University of Pisa

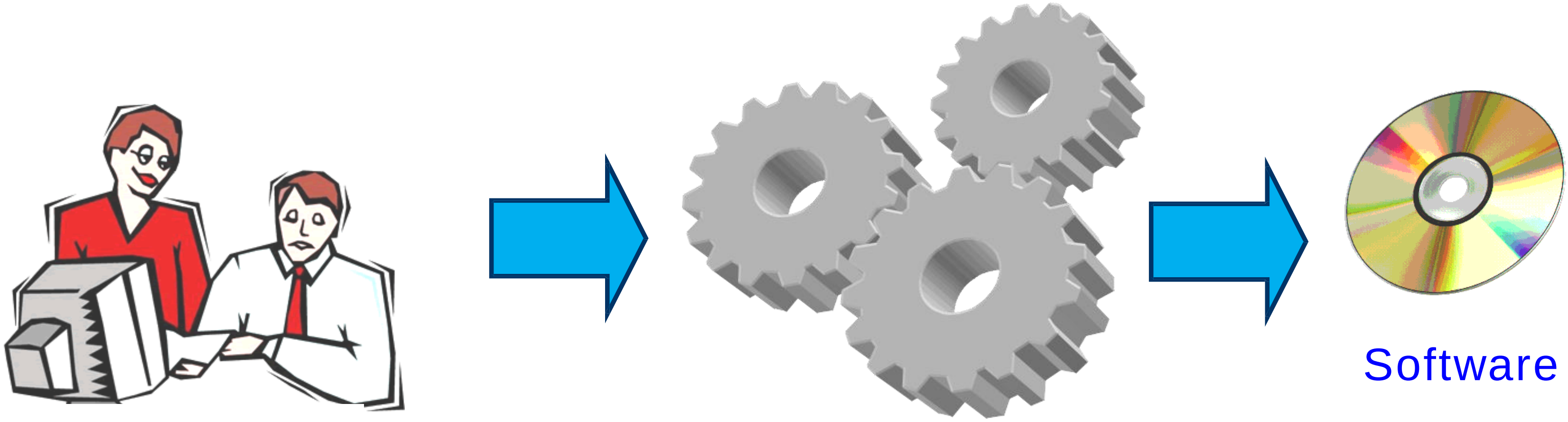
Computer Engineering – Software Systems Engineering

UML and Unified Process

Mario G. C. A. Cimino – Antonio Luca Alfeo

1. The Unified Process

✓ A Software Development Process (**SDP**) defines the *who*, *what*, *when*, and *how* of developing software. The Unified Process (**UP**) is an industry standard SDP from the authors of the **UML** (Unified Modeling Language).



business requirements

Unified Process

Software

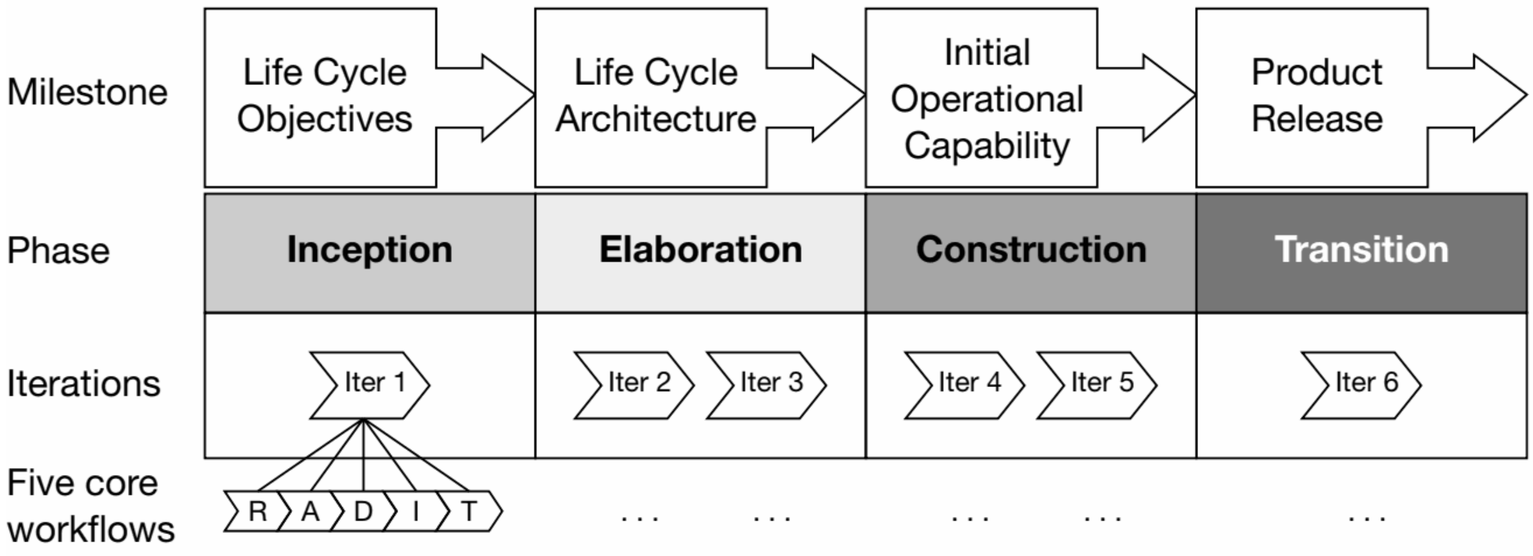
✓ UP is ***iterative*** and ***incremental***: a large software development project is broken down into smaller "mini projects" called ***iterations***. Each iteration generates a more complete version of the final system. The difference between two consecutive versions is called ***increment***.

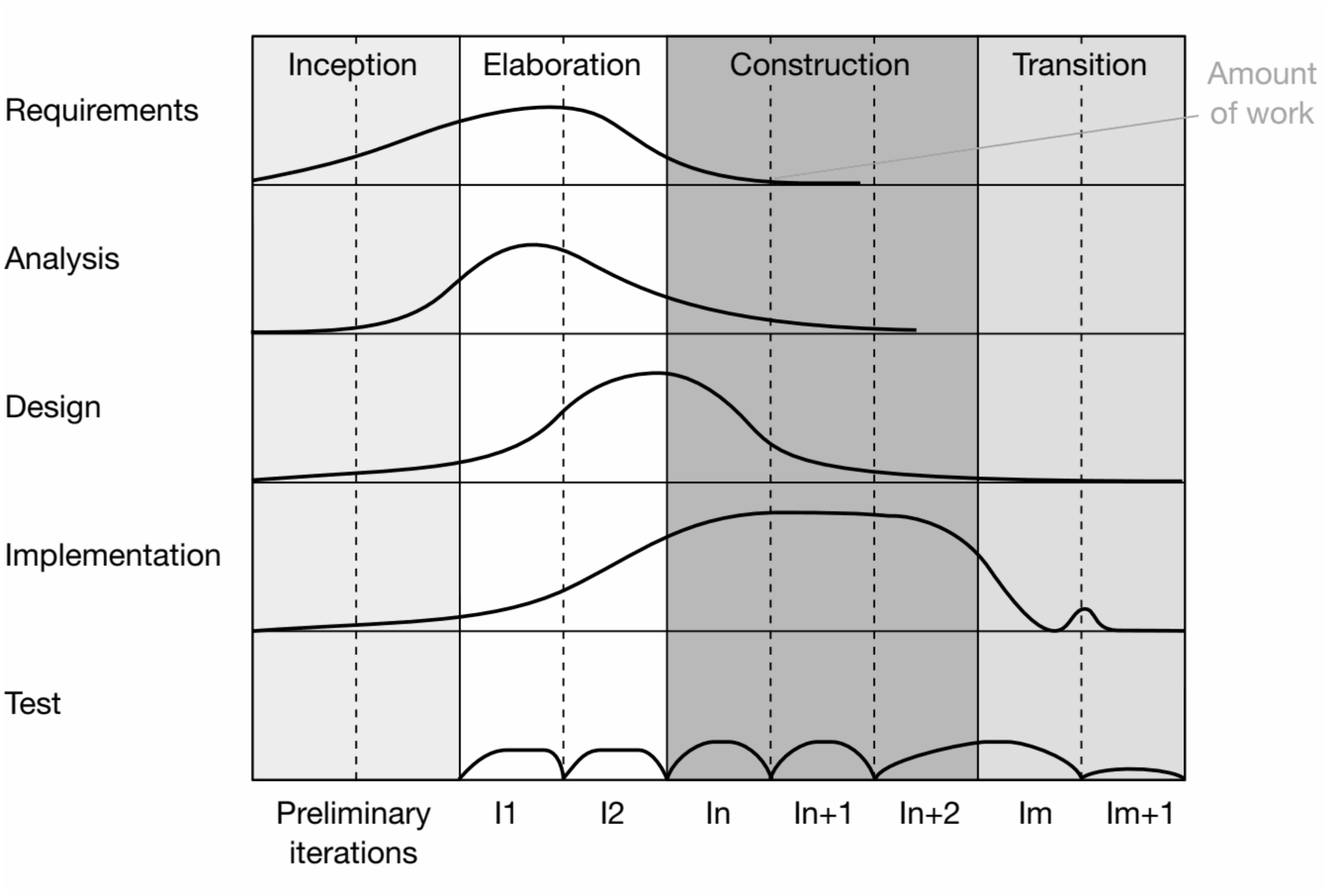
✓ Each iteration is made by five *core workflows*, with different emphasis:

- **R: requirements** – capturing what the system should do;
- **A: analysis** – refining and structuring the requirements;
- **D: design** – realizing the requirements in system architecture;
- **I: implementation** – building the software;
- **T: test** – verifying that the implementation works as desired.

✓ In a team work, it is often convenient to schedule iterations in parallel, according to dependencies between the artifacts of each iteration.

✓ UP consists of a sequence of four *phases*, terminating with related milestones:



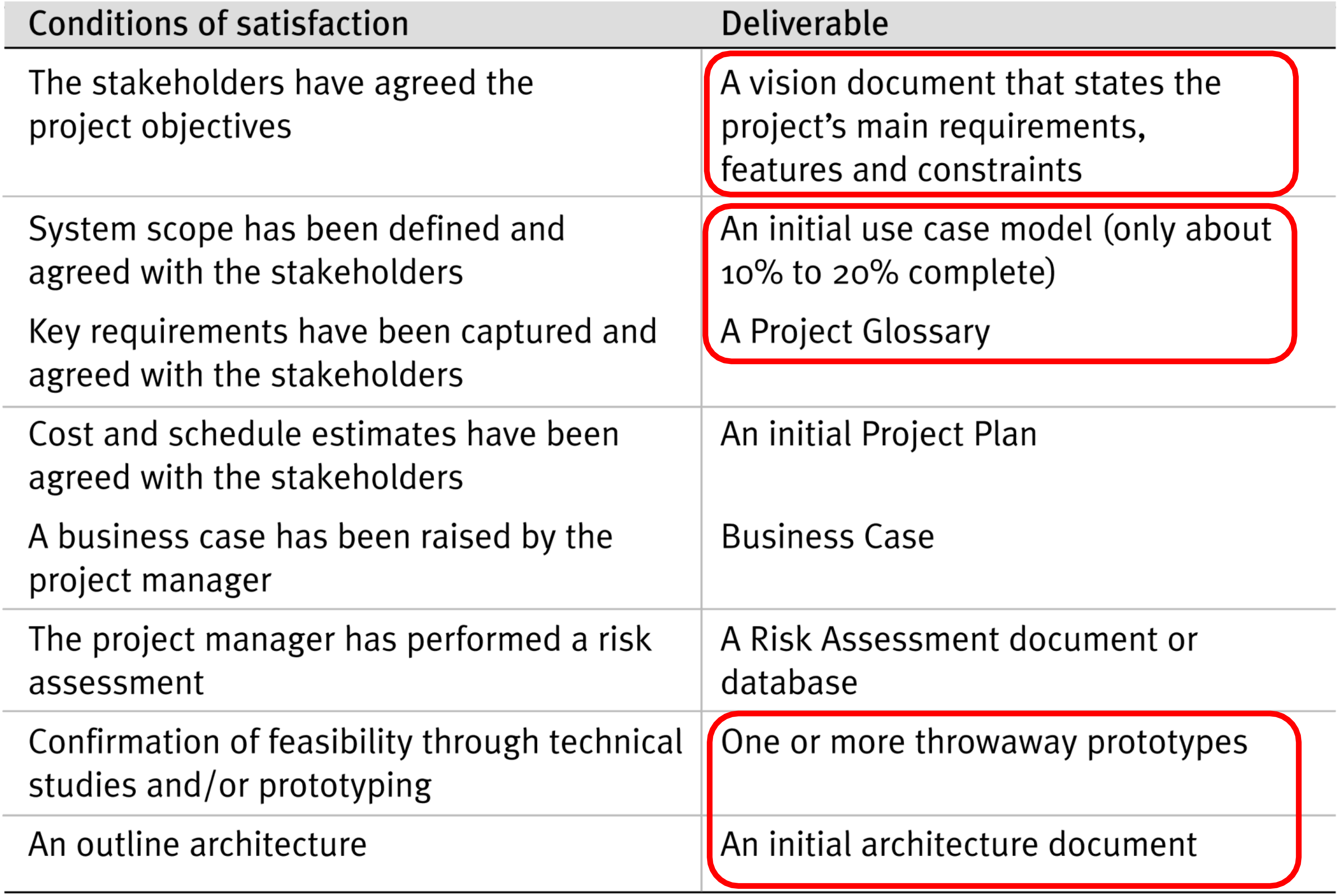


UP: core workflows versus phases

✓ **Inception**: most of the work in early requirements and analysis
Elaboration: the emphasis on requirements and analysis and some design
Construction: mostly design and implementation, with related testing
Transition: residual implementation and test

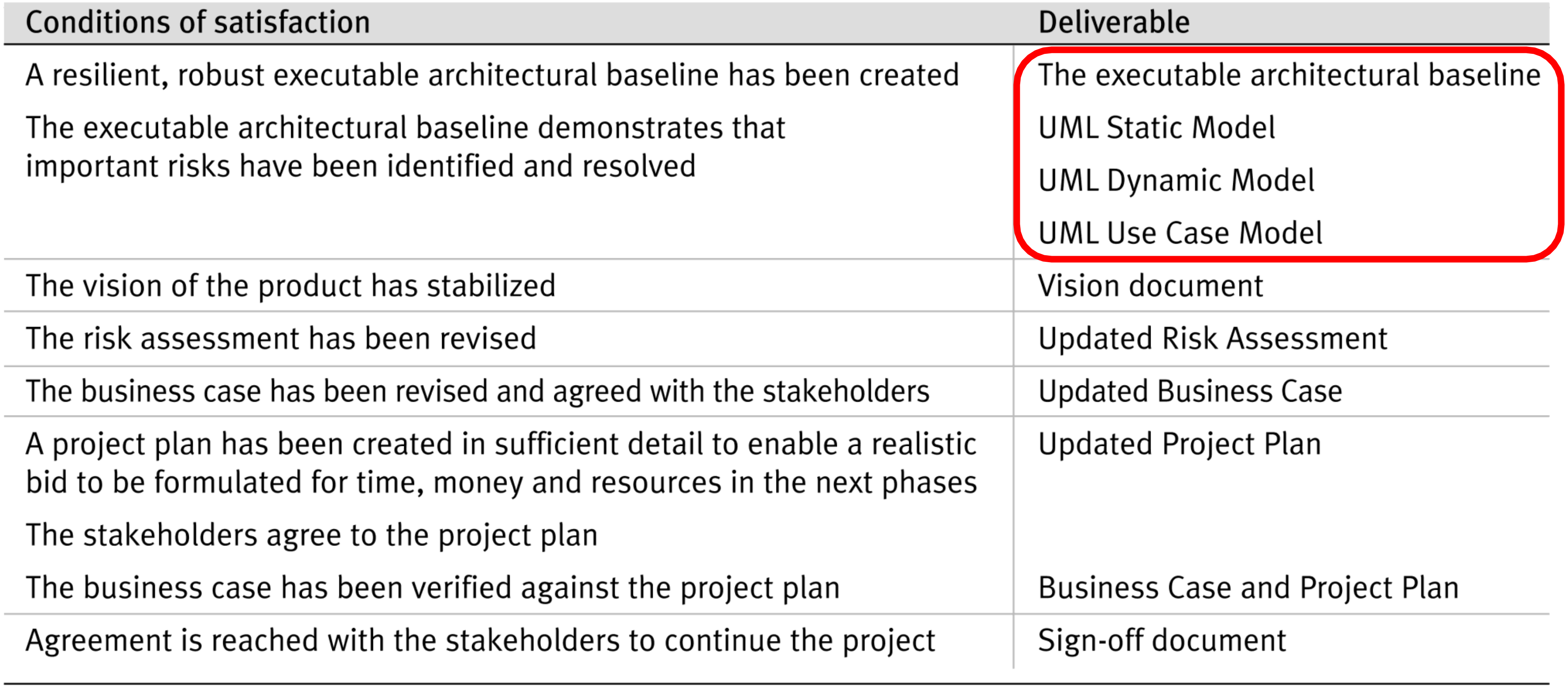
PHASE	**GOALS**	**FOCUS**	**MILESTONE**
Inception	●*capturing essential requirements* to help scope the system ●feasibility: technical prototype to validate technology, proof of concept to validate business requirements ●business case to demonstrate that the project will deliver quantifiable business benefit	●*requirements and analysis workflows* ●some design and implementation, to build technical prototype or proof of concept ● no testing – throwaway prototype	●Life Cycle Objectives (requirements/ features/constraints, initial use cases) → see conditions and deliverable table
Elaboration	● *create executable architectural baseline* ●*capture use cases* to 80% functional requirements; ● refine the Risk Assessment; ●define quality attributes (defect discovery rates, acceptable defect densities, etc.); ● create detailed plan for construction; ●formulate a bid that includes resources, time, equipment, staff and cost.	●*requirements, analysis and design workflows* ●implementation: build the initial operational capability ●test the initial operational capability (alpha test, internal)	●Life Cycle Architecture
Construction	●*complete requirements, analysis and design* ● move from architectural baseline to the *final system* ● maintain the system architecture integrity	● *implementation and testing* ●build the Initial Operational capability ●test the Initial Operational Capability	●Initial Operational Capability (software system is finished for beta testing in productive environment)
Transition	●starts after beta testing is completed and the system is finally deployed ●correct defects, prepare the user site for the new software; ● create user manuals and other documentation; provide user consultancy; ● conduct a post project review	● *no requirements, analysis* ●*finish implementation and complete test workflows* ●modify design if problems arise in beta testing ●user acceptance testing (user community)	●Product Release (the product is accepted into the user community)

Inception: conditions to attain for the Life Cycle Objectives



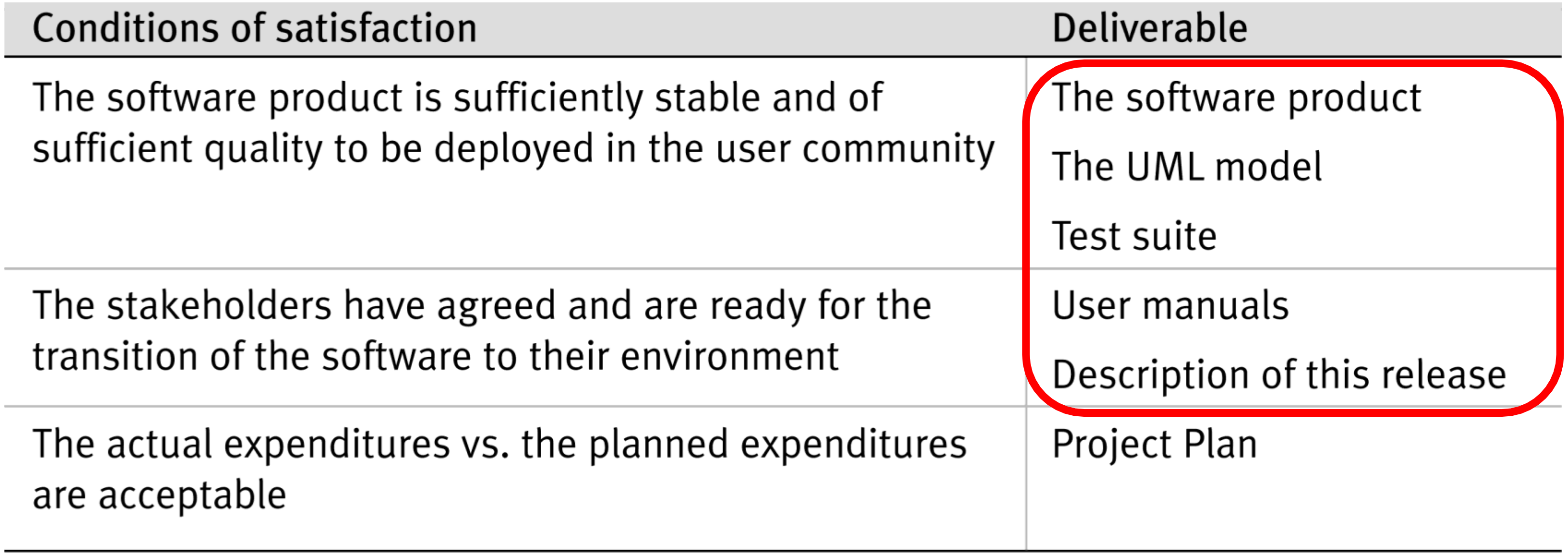
Conditions of satisfaction	Deliverable
The stakeholders have agreed the project objectives	A vision document that states the project's main requirements, features and constraints
System scope has been defined and agreed with the stakeholders	An initial use case model (only about 10% to 20% complete)
Key requirements have been captured and agreed with the stakeholders	A Project Glossary
Cost and schedule estimates have been agreed with the stakeholders	An initial Project Plan
A business case has been raised by the project manager	Business Case
The project manager has performed a risk assessment	A Risk Assessment document or database
Confirmation of feasibility through technical studies and/or prototyping	One or more throwaway prototypes
An outline architecture	An initial architecture document

Elaboration: conditions to attain for the Life Cycle Architecture



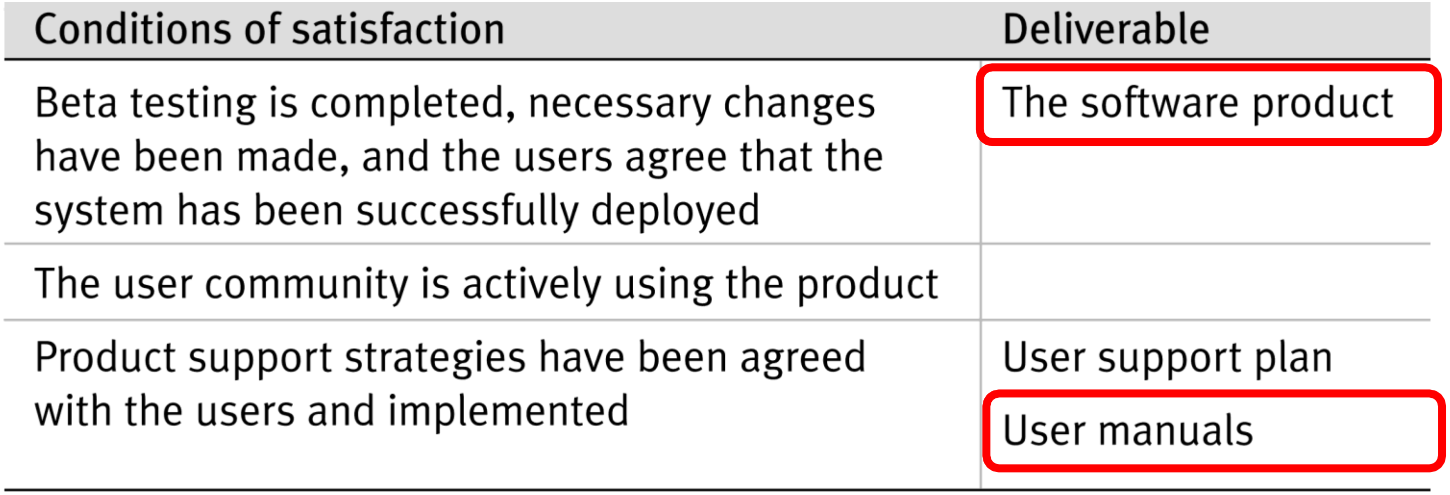
Conditions of satisfaction	Deliverable
A resilient, robust executable architectural baseline has been created	The executable architectural baseline
The executable architectural baseline demonstrates that important risks have been identified and resolved	UML Static Model
	UML Dynamic Model
	UML Use Case Model
The vision of the product has stabilized	Vision document
The risk assessment has been revised	Updated Risk Assessment
The business case has been revised and agreed with the stakeholders	Updated Business Case
A project plan has been created in sufficient detail to enable a realistic bid to be formulated for time, money and resources in the next phases	Updated Project Plan
The stakeholders agree to the project plan	
The business case has been verified against the project plan	Business Case and Project Plan
Agreement is reached with the stakeholders to continue the project	Sign-off document

Construction: conditions to attain for the Initial Operational Capability



Conditions of satisfaction	Deliverable
The software product is sufficiently stable and of sufficient quality to be deployed in the user community	The software product The UML model Test suite
The stakeholders have agreed and are ready for the transition of the software to their environment	User manuals Description of this release
The actual expenditures vs. the planned expenditures are acceptable	Project Plan

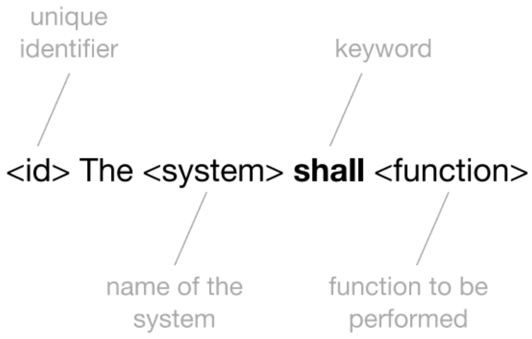
Transition: conditions to attain for the Product Release



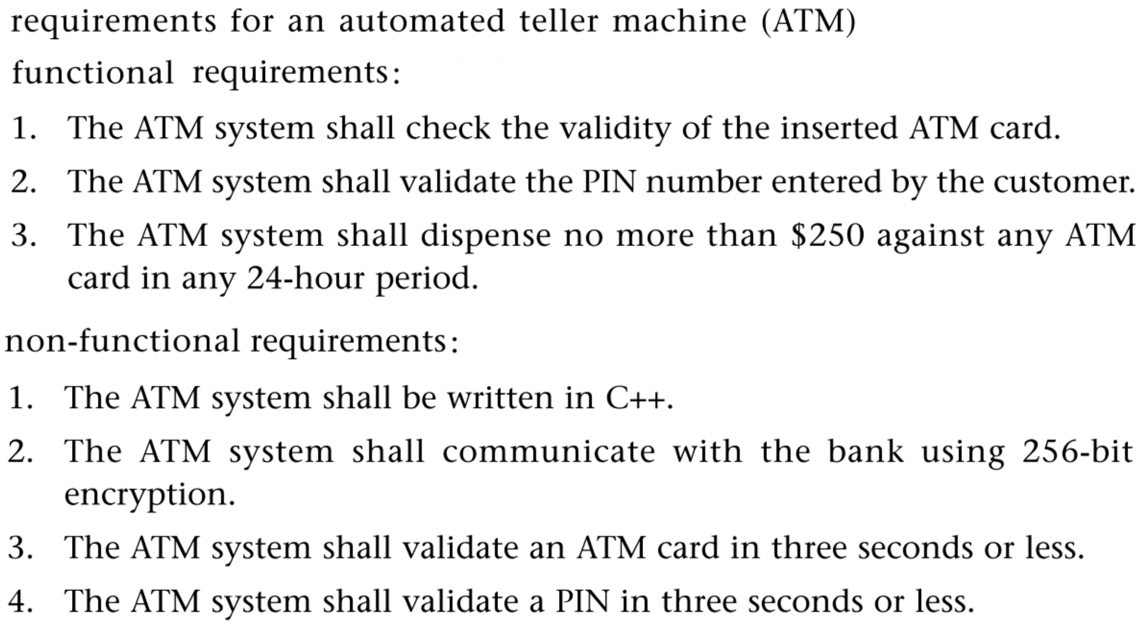
Conditions of satisfaction	Deliverable
Beta testing is completed, necessary changes have been made, and the users agree that the system has been successfully deployed	The software product
The user community is actively using the product	
Product support strategies have been agreed with the users and implemented	User support plan User manuals

2. The requirements workflow

- ✓ Requirements: statements on *what* the system should do (functional) and *how* it should do it (constraints, properties, non-functional)
- ✓ Well-formed requirements:



- ✓ Example:



requirements for an automated teller machine (ATM)

functional requirements:

1. The ATM system shall check the validity of the inserted ATM card.
2. The ATM system shall validate the PIN number entered by the customer.
3. The ATM system shall dispense no more than $250 against any ATM card in any 24-hour period.

non-functional requirements:

1. The ATM system shall be written in C++.
2. The ATM system shall communicate with the bank using 256-bit encryption.
3. The ATM system shall validate an ATM card in three seconds or less.
4. The ATM system shall validate a PIN in three seconds or less.

✓ Questions helping to identify actors:

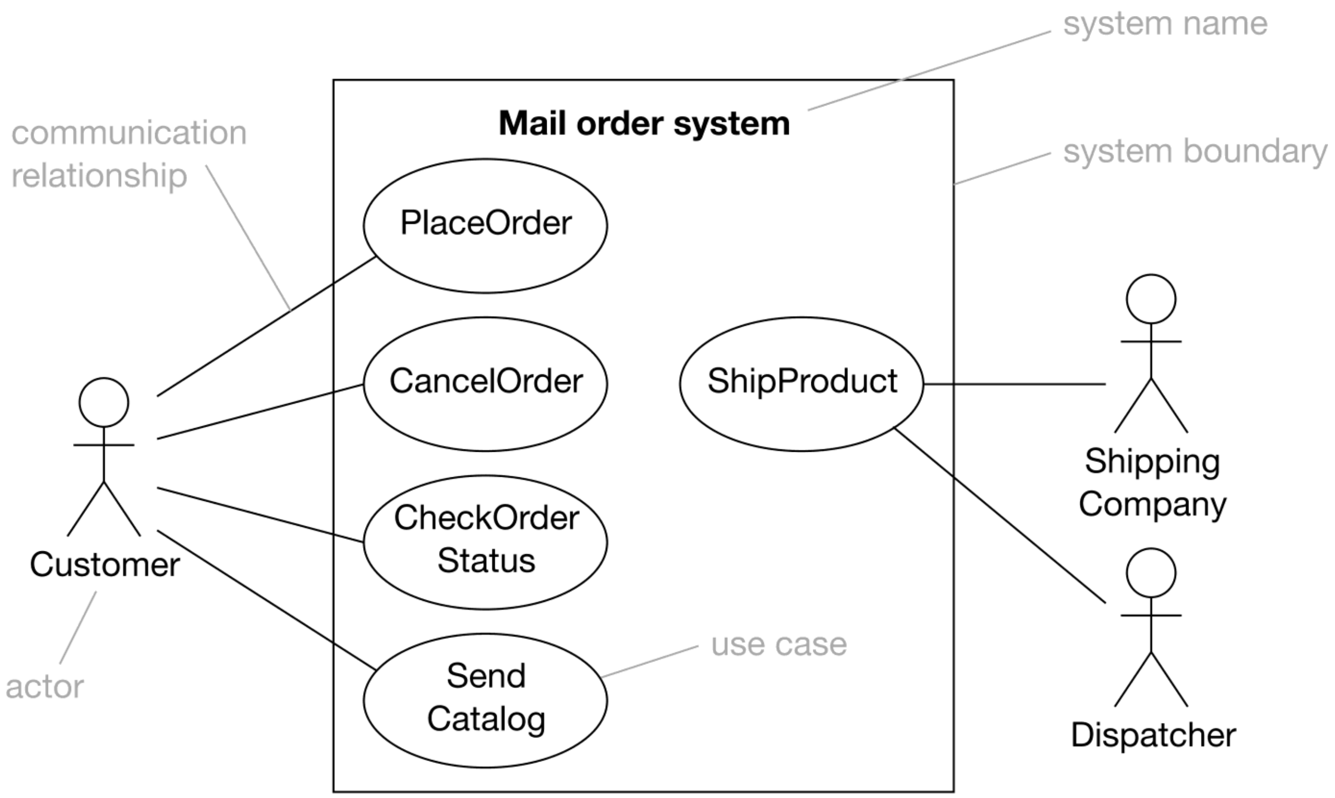
> To find actors ask:
> “Who or what uses or interacts with the system?”

- Who or what uses the system?
- What roles do they play in the interaction?
- Who installs the system?
- Who starts and shuts down the system?
- Who maintains the system?
- What other systems interact with this system?
- Who gets and provides information to the system?
- Does anything happen at a fixed time?

✓ Questions helping to identify use cases:

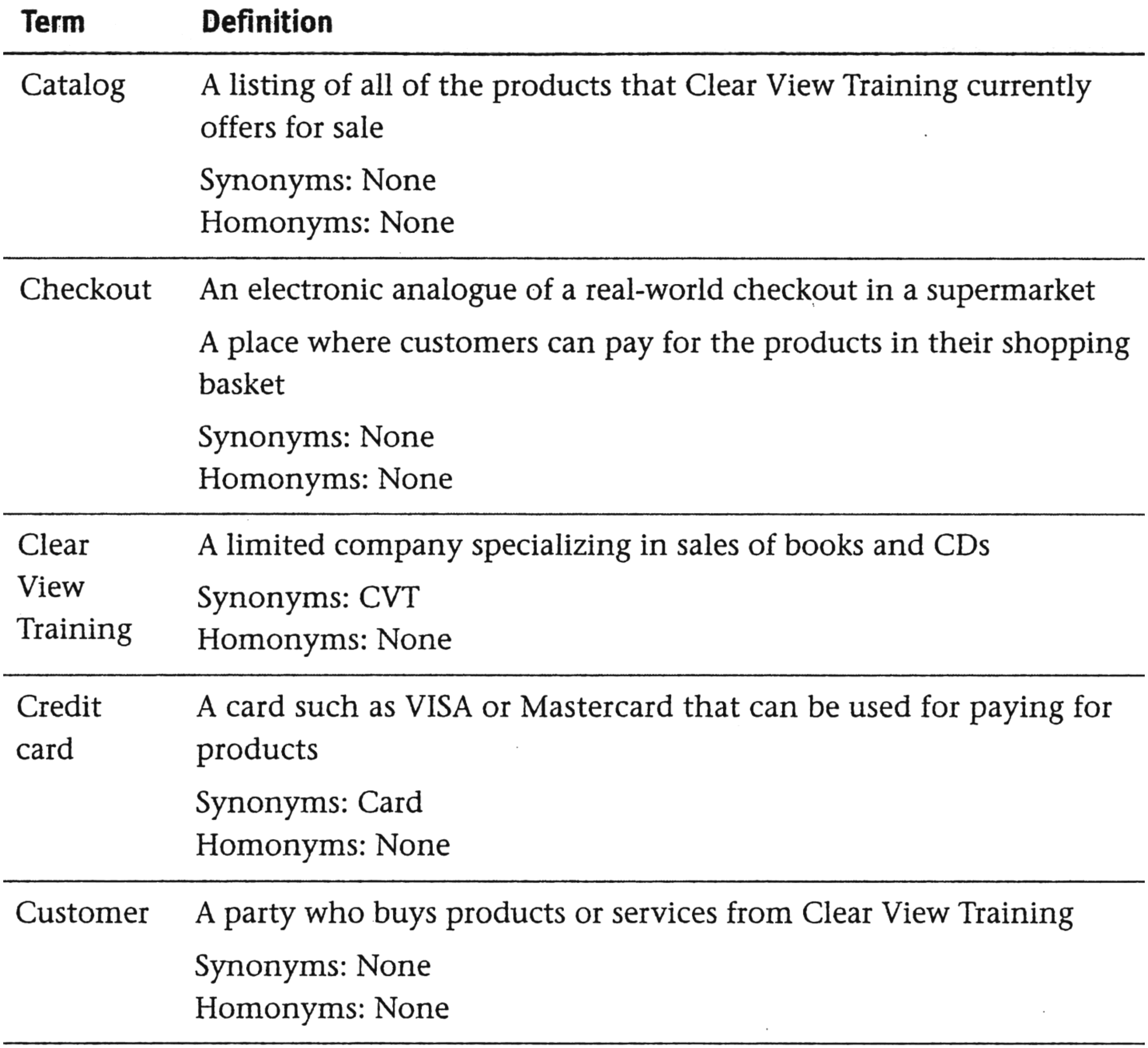
- What functions will a specific actor want from the system?
- Does the system store and retrieve information? If so, which actors trigger this behavior?
- Are any actors notified when the system changes state?
- Are there any external events that affect the system? What notifies the system about those events?

- ✓ The UML use case diagram:



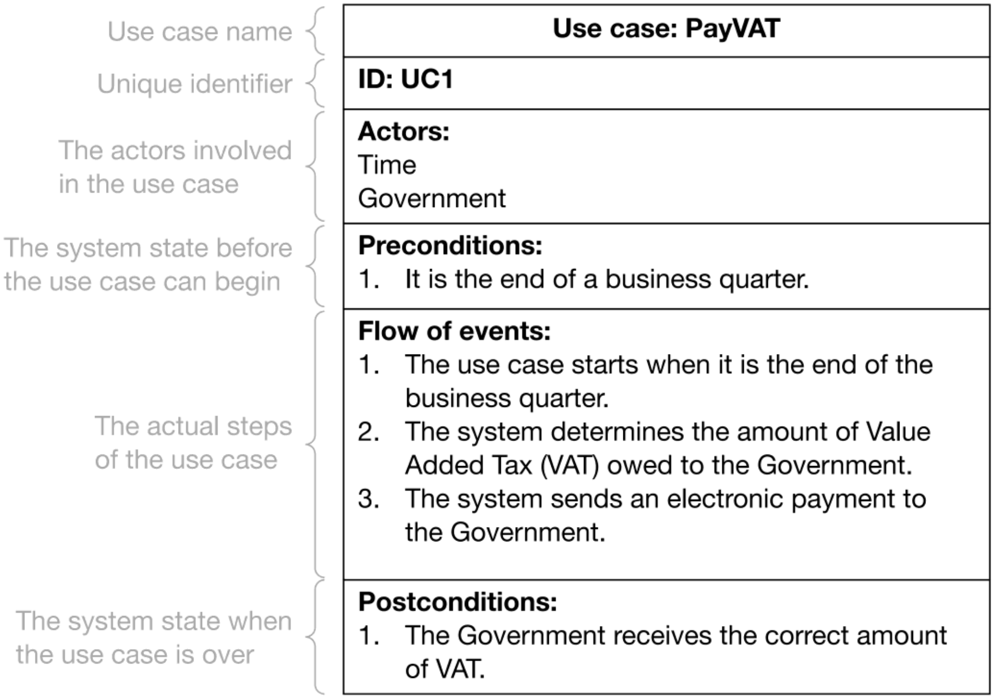
- ✓ The *project glossary*: is a list of *key business terms*, related *definitions*, *synonyms* (different terms for the same concept → use a unique preferred term) and *homonyms* (same term for different concepts → qualify such terms)

- ✓ Use case specification: (a) pre/post-conditions, things that must be true before/after the start/end of the use case; (b) flow of events, steps in the use case.



Term	Definition
Catalog	A listing of all of the products that Clear View Training currently offers for sale Synonyms: None Homonyms: None
Checkout	An electronic analogue of a real-world checkout in a supermarket A place where customers can pay for the products in their shopping basket Synonyms: None Homonyms: None
Clear View Training	A limited company specializing in sales of books and CDs Synonyms: CVT Homonyms: None
Credit card	A card such as VISA or Mastercard that can be used for paying for products Synonyms: Card Homonyms: None
Customer	A party who buys products or services from Clear View Training Synonyms: None Homonyms: None

✓ Anatomy of a detailed use case:



Use case name

Use case: PayVAT

Unique identifier

ID: UC1

The actors involved in the use case

Actors:
Time
Government

The system state before the use case can begin

Preconditions:

1. It is the end of a business quarter.

The actual steps of the use case

Flow of events:

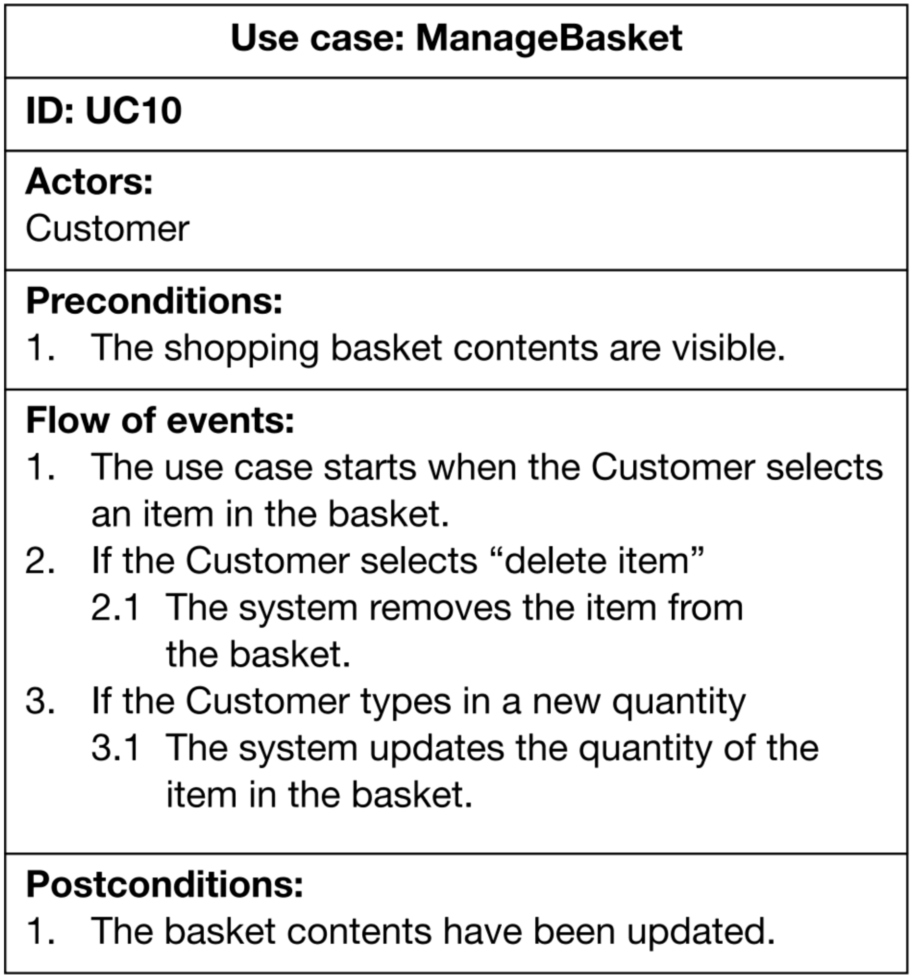
1. The use case starts when it is the end of the business quarter.
2. The system determines the amount of Value Added Tax (VAT) owed to the Government.
3. The system sends an electronic payment to the Government.

The system state when the use case is over

Postconditions:

1. The Government receives the correct amount of VAT.

- ✓ Branching within a flow: **IF**



Use case: ManageBasket

ID: UC10

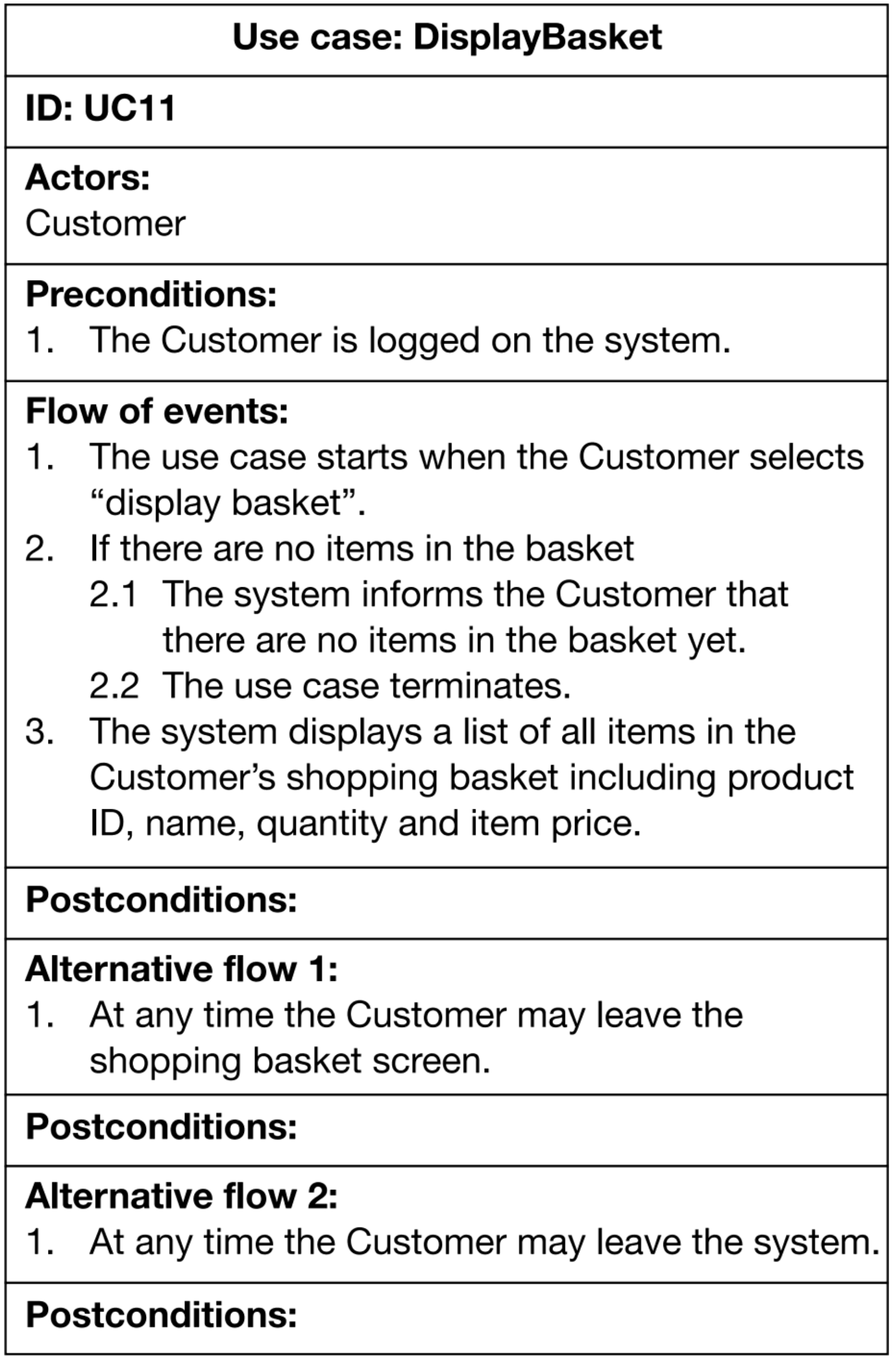
Actors:
Customer

Preconditions:
1. The shopping basket contents are visible.

Flow of events:
1. The use case starts when the Customer selects an item in the basket.
2. If the Customer selects "delete item"
   2.1 The system removes the item from the basket.
3. If the Customer types in a new quantity
   3.1 The system updates the quantity of the item in the basket.

Postconditions:
1. The basket contents have been updated.

- ✓ Alternative flows: e.g. for things happening under conditions potentially occurring at any step of the use case



Use case: DisplayBasket

ID: UC11

Actors:
Customer

Preconditions:

1. The Customer is logged on the system.

Flow of events:

1. The use case starts when the Customer selects "display basket".
2. If there are no items in the basket
   - 2.1 The system informs the Customer that there are no items in the basket yet.
   - 2.2 The use case terminates.
3. The system displays a list of all items in the Customer's shopping basket including product ID, name, quantity and item price.

Postconditions:

Alternative flow 1:

1. At any time the Customer may leave the shopping basket screen.

Postconditions:

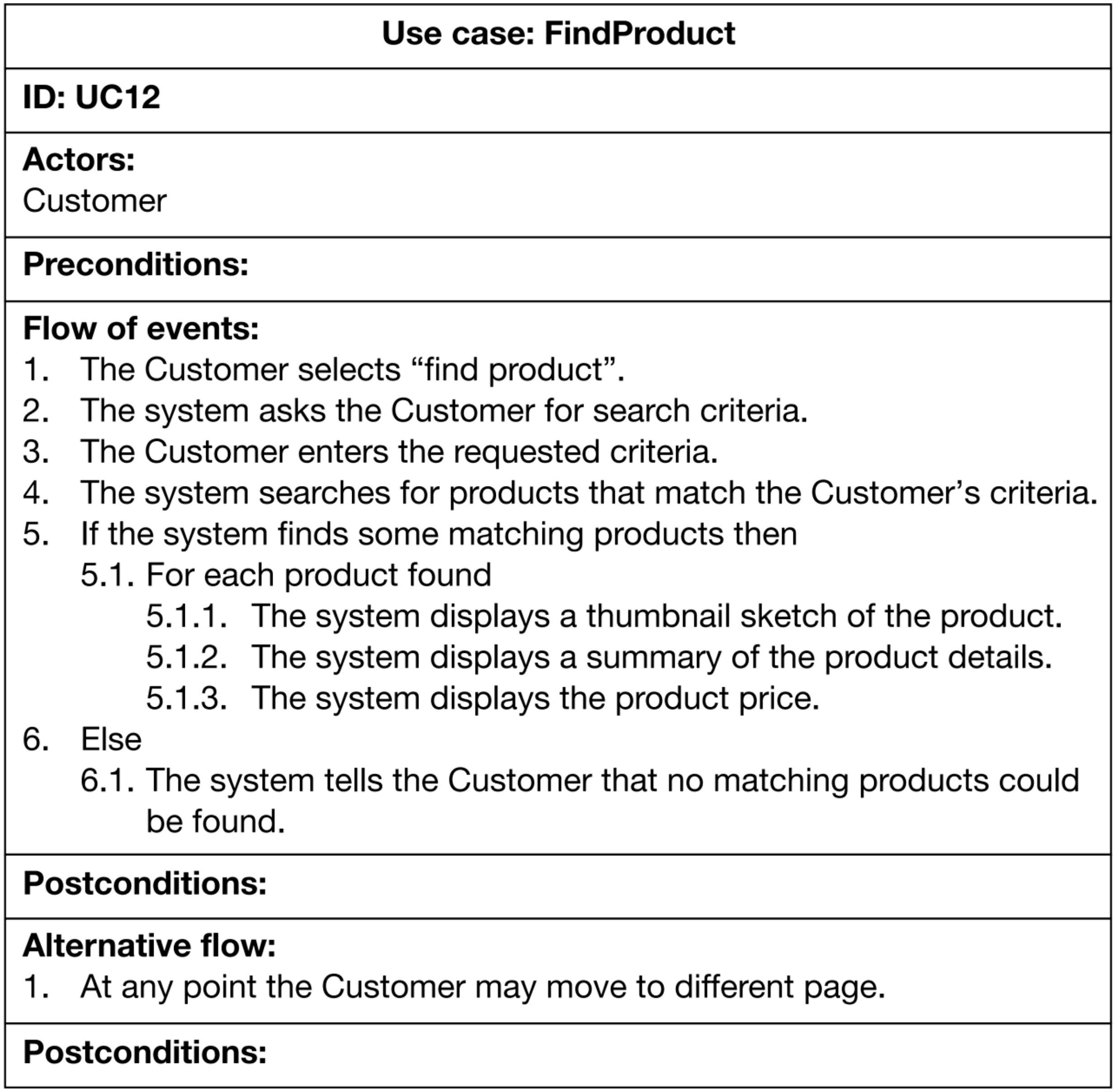
Alternative flow 2:

1. At any time the Customer may leave the system.

Postconditions:

✓ Repetition within a flow: **FOR**

```
n. For (iteration expression)
        n.1. Do something
        n.2. Do something else
        n.3. ...
n+1.
```



Use case: FindProduct

ID: UC12

Actors:
Customer

Preconditions:

Flow of events:

1. The Customer selects “find product”.
2. The system asks the Customer for search criteria.
3. The Customer enters the requested criteria.
4. The system searches for products that match the Customer’s criteria.
5. If the system finds some matching products then
    5.1. For each product found
        5.1.1. The system displays a thumbnail sketch of the product.
        5.1.2. The system displays a summary of the product details.
        5.1.3. The system displays the product price.
6. Else
    6.1. The system tells the Customer that no matching products could be found.

Postconditions:

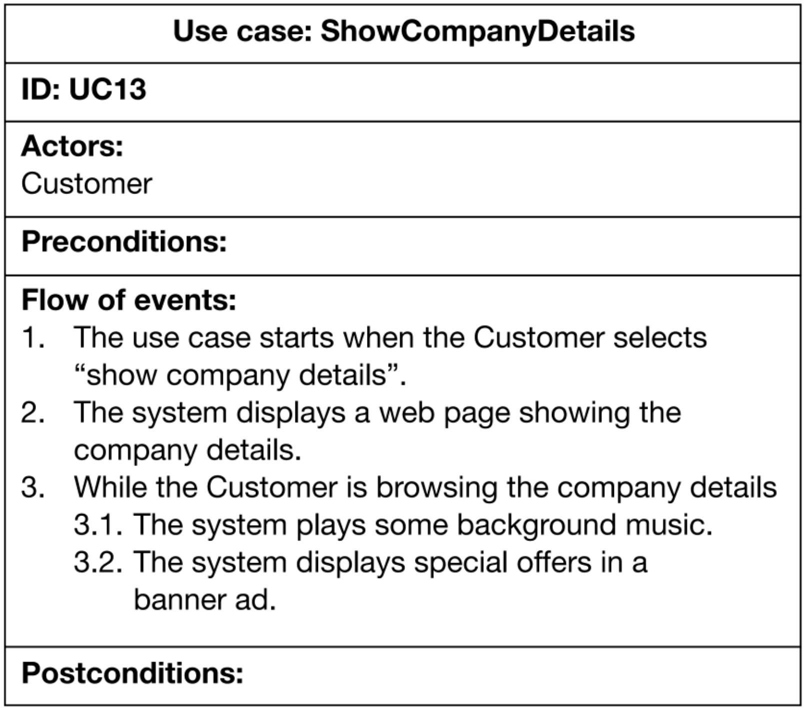
Alternative flow:

1. At any point the Customer may move to different page.

Postconditions:

✓ Repetition within a flow: **WHILE**

```
n. While (Boolean condition)
        n.1. Do something
        n.2. Do something else
        n.3. ...
n+1.
```



Use case: ShowCompanyDetails

ID: UC13

Actors:
Customer

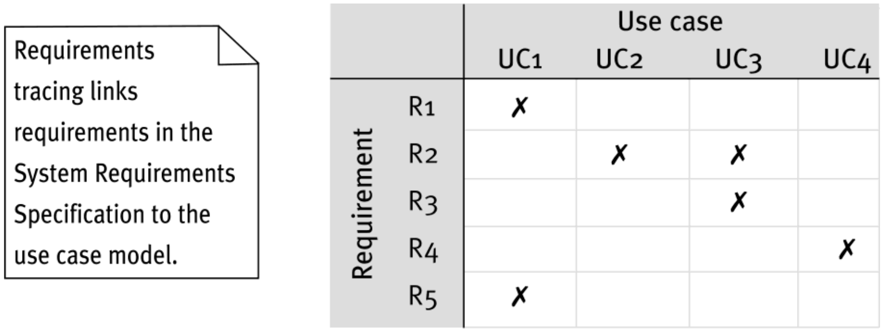
Preconditions:

Flow of events:

1. The use case starts when the Customer selects “show company details”.
2. The system displays a web page showing the company details.
3. While the Customer is browsing the company details
   3.1. The system plays some background music.
   3.2. The system displays special offers in a banner ad.

Postconditions:

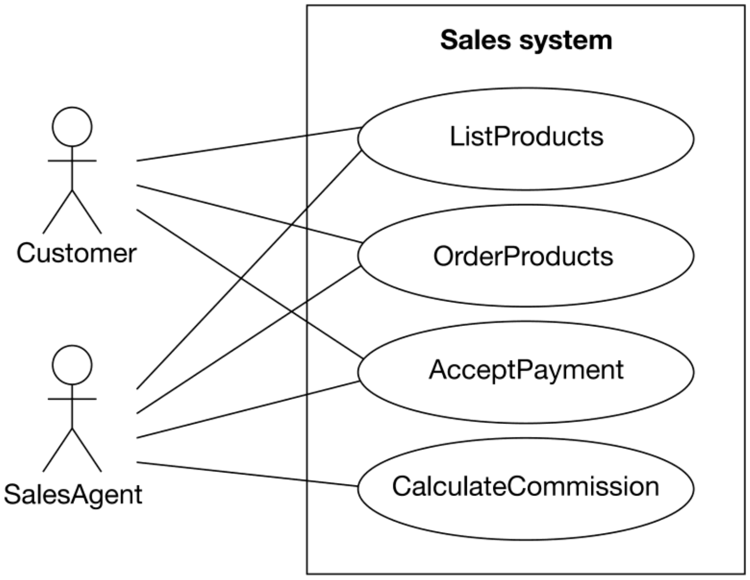
✓ Requirements tracing: many-to-many relationship between requirements and use cases, how to discover missing use cases or missing requirements.

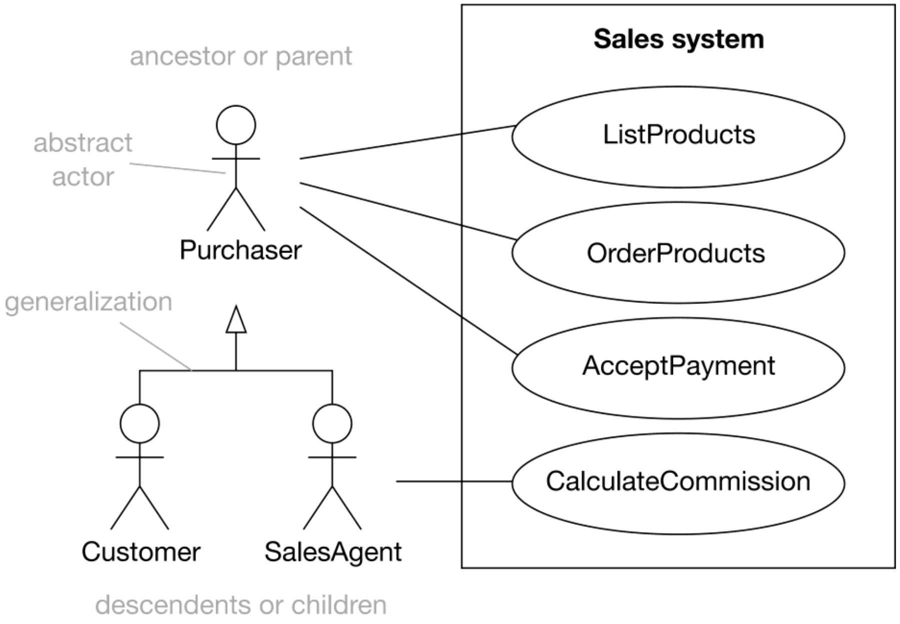


Requirements tracing links requirements in the System Requirements Specification to the use case model.

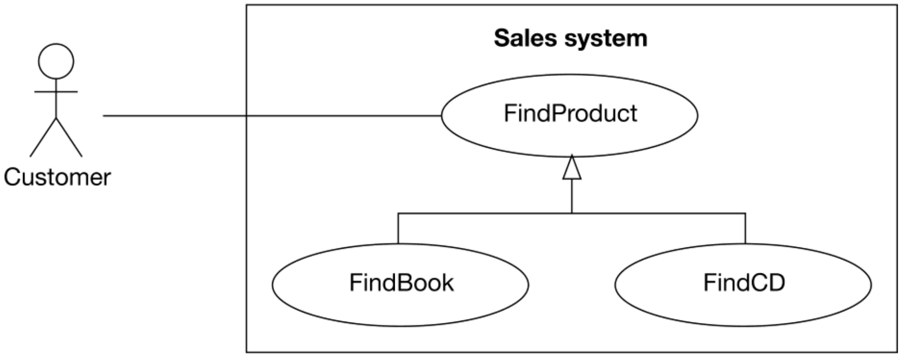
		Use case			
		UC1	UC2	UC3	UC4
Requirement	R1	✗			
	R2		✗	✗	
	R3			✗	
	R4				✗
	R5	✗			

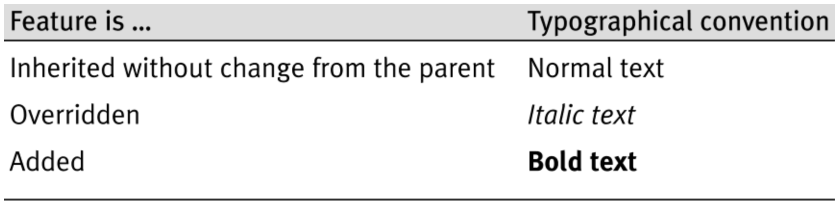
✓ *Actor generalization*: the descendent actors inherit the roles and relationships to use cases held by the parent actor



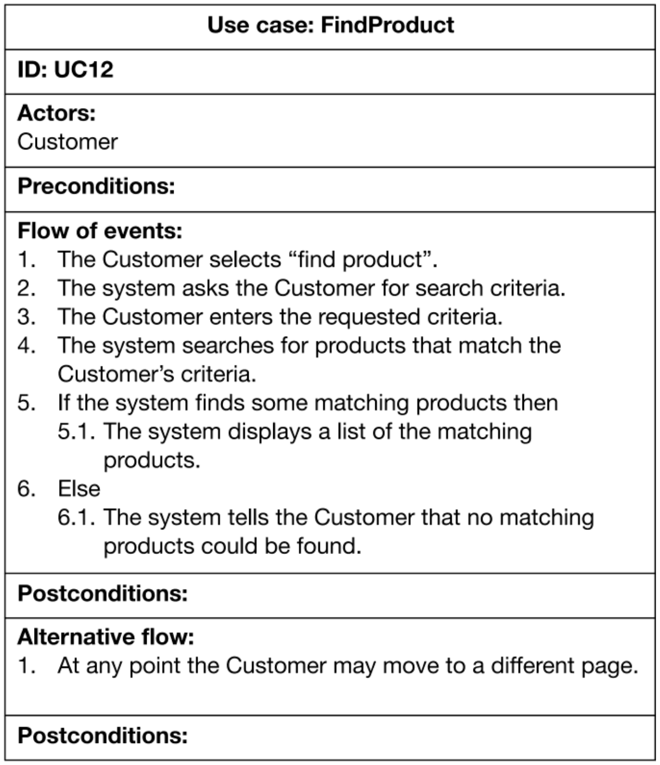


✓ *Use case generalization*: the child use case inherits features from the parent use case, can add or change (override) inherited features (pre/post condition, steps in flow…)





Feature is ...	Typographical convention
Inherited without change from the parent	Normal text
Overridden	*Italic text*
Added	**Bold text**



Use case: FindProduct

ID: UC12

Actors:
Customer

Preconditions:

Flow of events:

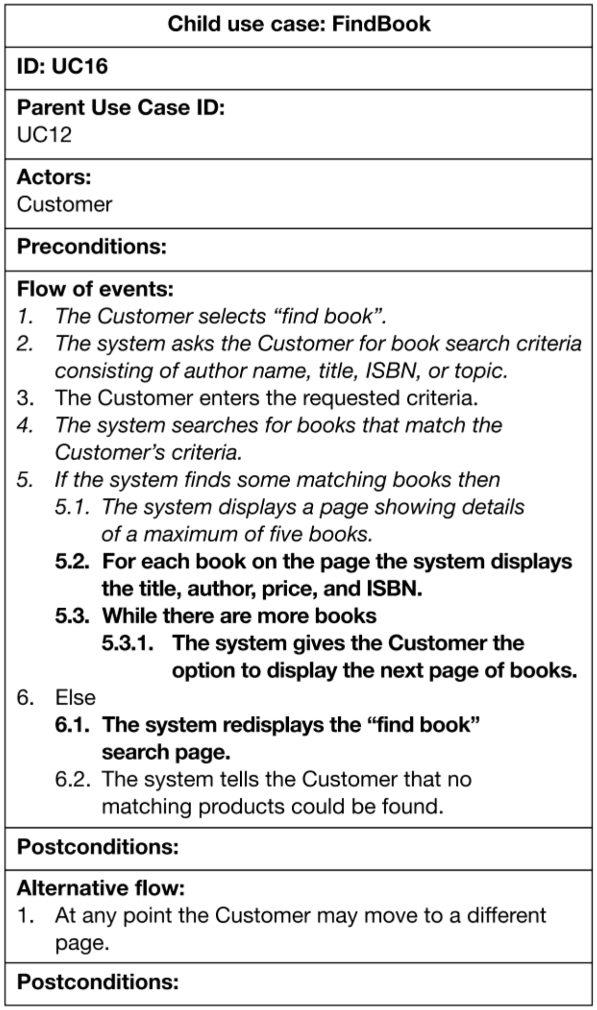
1. The Customer selects “find product”.
2. The system asks the Customer for search criteria.
3. The Customer enters the requested criteria.
4. The system searches for products that match the Customer’s criteria.
5. If the system finds some matching products then
   - 5.1. The system displays a list of the matching products.
6. Else
   - 6.1. The system tells the Customer that no matching products could be found.

Postconditions:

Alternative flow:

1. At any point the Customer may move to a different page.

Postconditions:



Child use case: FindBook

ID: UC16

Parent Use Case ID:
UC12

Actors:
Customer

Preconditions:

Flow of events:

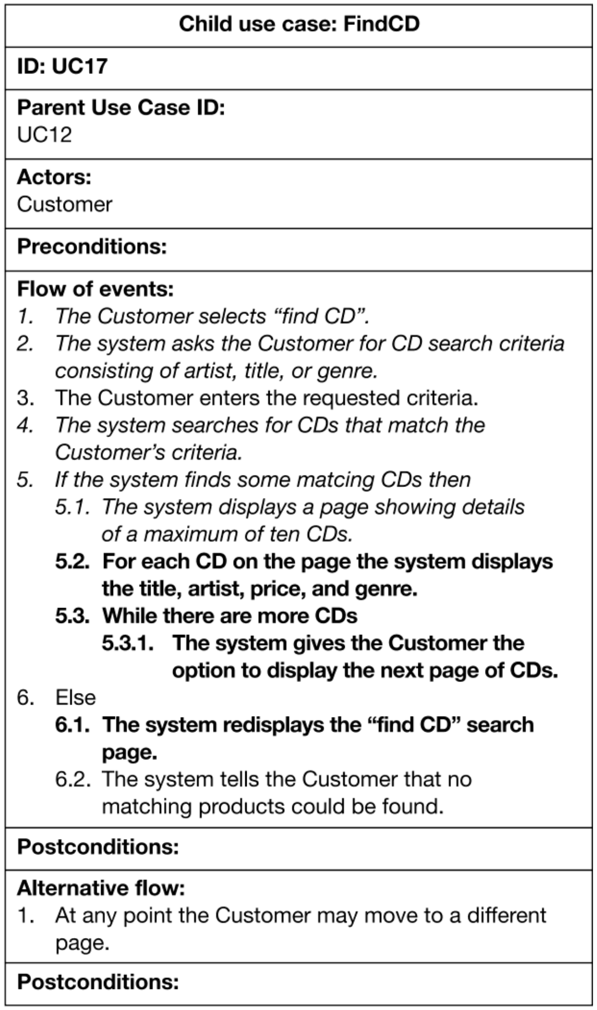
1. *The Customer selects “find book”.*
2. *The system asks the Customer for book search criteria consisting of author name, title, ISBN, or topic.*
3. The Customer enters the requested criteria.
4. *The system searches for books that match the Customer’s criteria.*
5. *If the system finds some matching books then*
   - *5.1. The system displays a page showing details of a maximum of five books.*
   - **5.2. For each book on the page the system displays the title, author, price, and ISBN.**
   - **5.3. While there are more books**
     - **5.3.1. The system gives the Customer the option to display the next page of books.**
6. Else
   - **6.1. The system redisplays the “find book” search page.**
   - 6.2. The system tells the Customer that no matching products could be found.

Postconditions:

Alternative flow:

1. At any point the Customer may move to a different page.

Postconditions:



Child use case: FindCD

ID: UC17

Parent Use Case ID:
UC12

Actors:
Customer

Preconditions:

Flow of events:

1. *The Customer selects “find CD”.*
2. *The system asks the Customer for CD search criteria consisting of artist, title, or genre.*
3. The Customer enters the requested criteria.
4. *The system searches for CDs that match the Customer’s criteria.*
5. *If the system finds some matcing CDs then*
   - *5.1. The system displays a page showing details of a maximum of ten CDs.*
   - **5.2. For each CD on the page the system displays the title, artist, price, and genre.**
   - **5.3. While there are more CDs**
     - **5.3.1. The system gives the Customer the option to display the next page of CDs.**
6. Else
   - **6.1. The system redisplays the “find CD” search page.**
   - 6.2. The system tells the Customer that no matching products could be found.

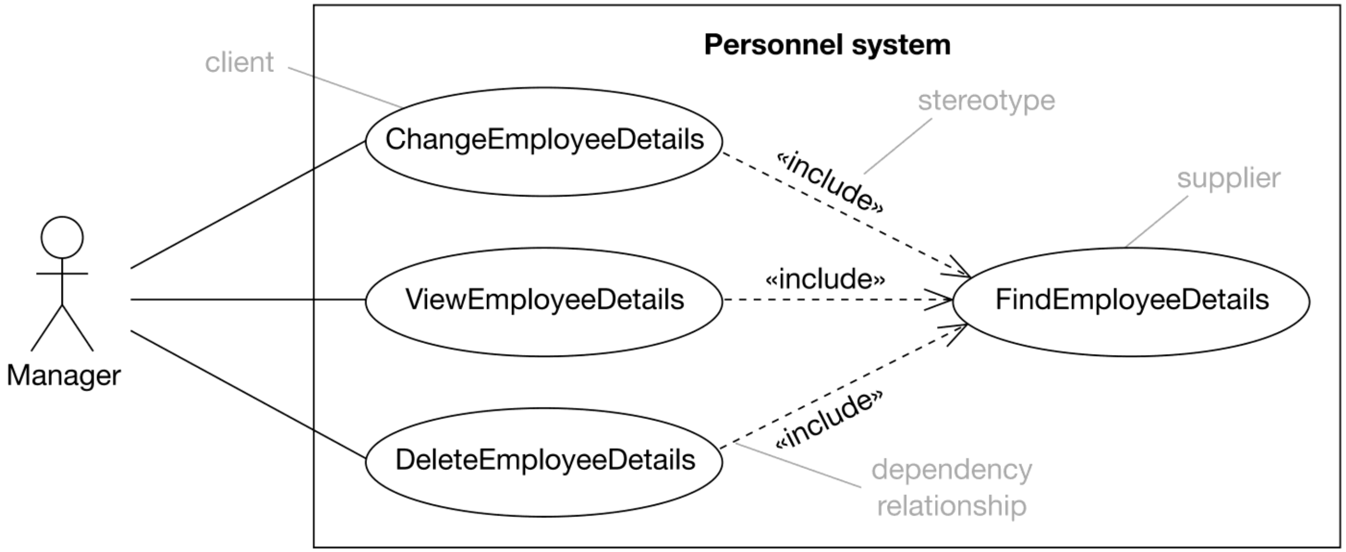
Postconditions:

Alternative flow:

1. At any point the Customer may move to a different page.

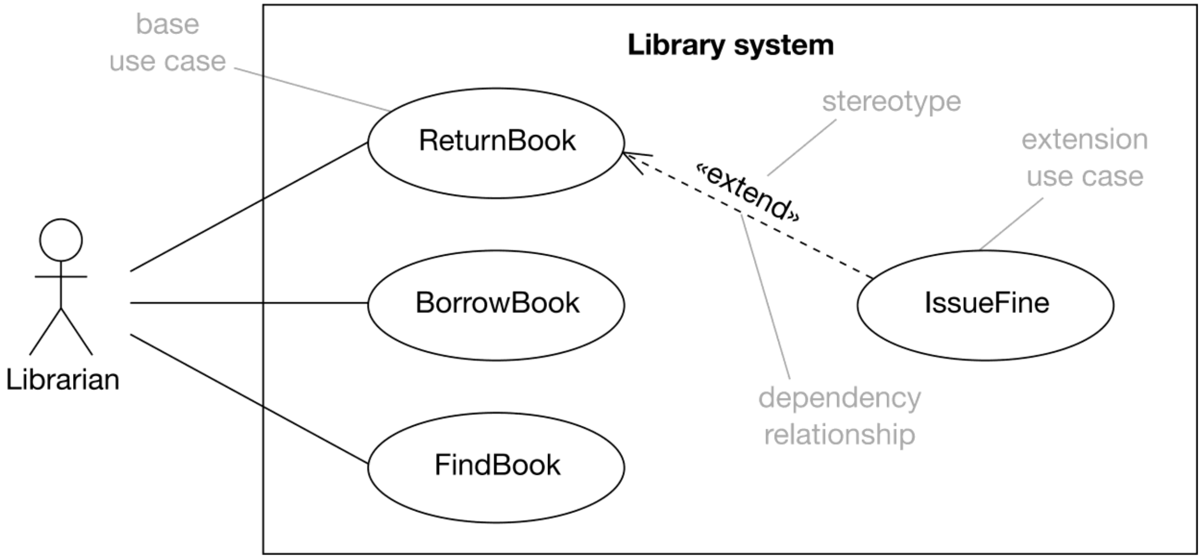
Postconditions:

✓ *The «include» relationship* between use cases includes the behavior of a supplier use case into the flow of a client use case. The client use case is not complete without all of its supplier use cases. The supplier use cases may or may not be complete (behavior fragment, it is not instantiable, it cannot be triggered directly by actors)

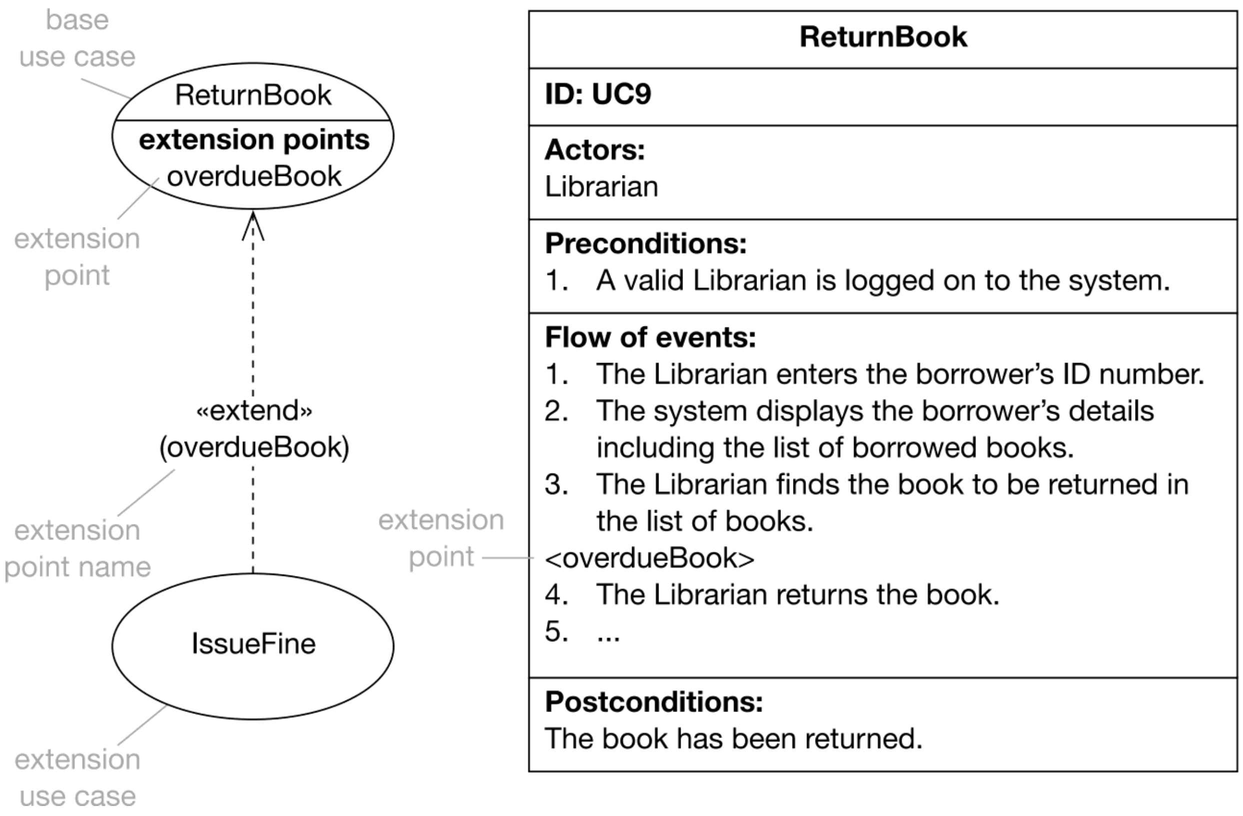


«include» factors out steps common to several use cases into a separate use case which is then included.

✓ *The «extend» relationship* between use cases adds new behavior to a base use case. The base use case is complete without its extensions (that usually are not complete).



- ✓ The extension points are added to an overlay on top of the flow of events, without effects on the numbering of the flow of events of the base use case.



ReturnBook

ID: UC9

Actors:
Librarian

Preconditions:
1. A valid Librarian is logged on to the system.

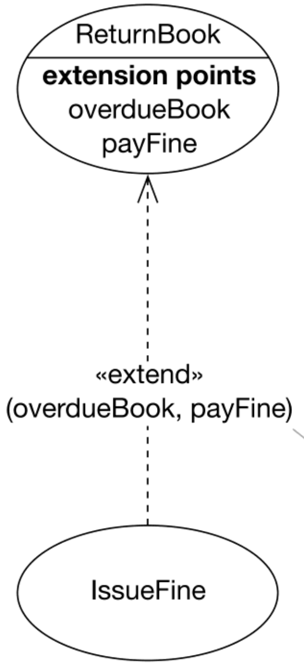
Flow of events:
1. The Librarian enters the borrower's ID number.
2. The system displays the borrower's details including the list of borrowed books.
3. The Librarian finds the book to be returned in the list of books.

<overdueBook>

4. The Librarian returns the book.
5. ...

Postconditions:
The book has been returned.

- ✓ Multiple *insertion segments* can be added.
- ✓ *Conditional extensions* are also possible. A condition is a Boolean expression.



the first insertion segment in IssueFine is inserted at <overdueBook> and the second insertion segment at <payFine>

Extension use case: IssueFine

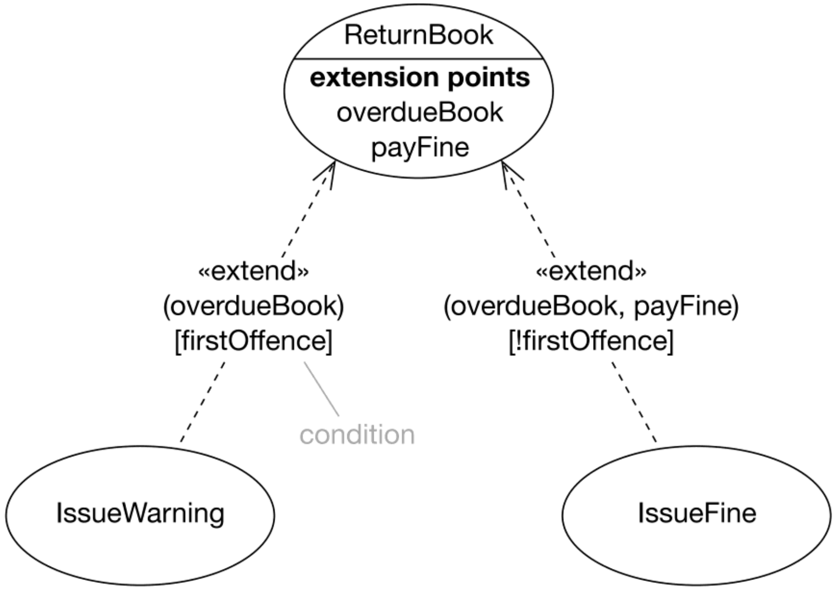
ID: UC10

Insertion segment 1:

1. The Librarian uses the system to record and print a fine.

Insertion segment 2:

1. If the borrower chooses to pay the total fine on the spot
    1.1. The Librarian accepts payment for the fine.
    1.2. ...

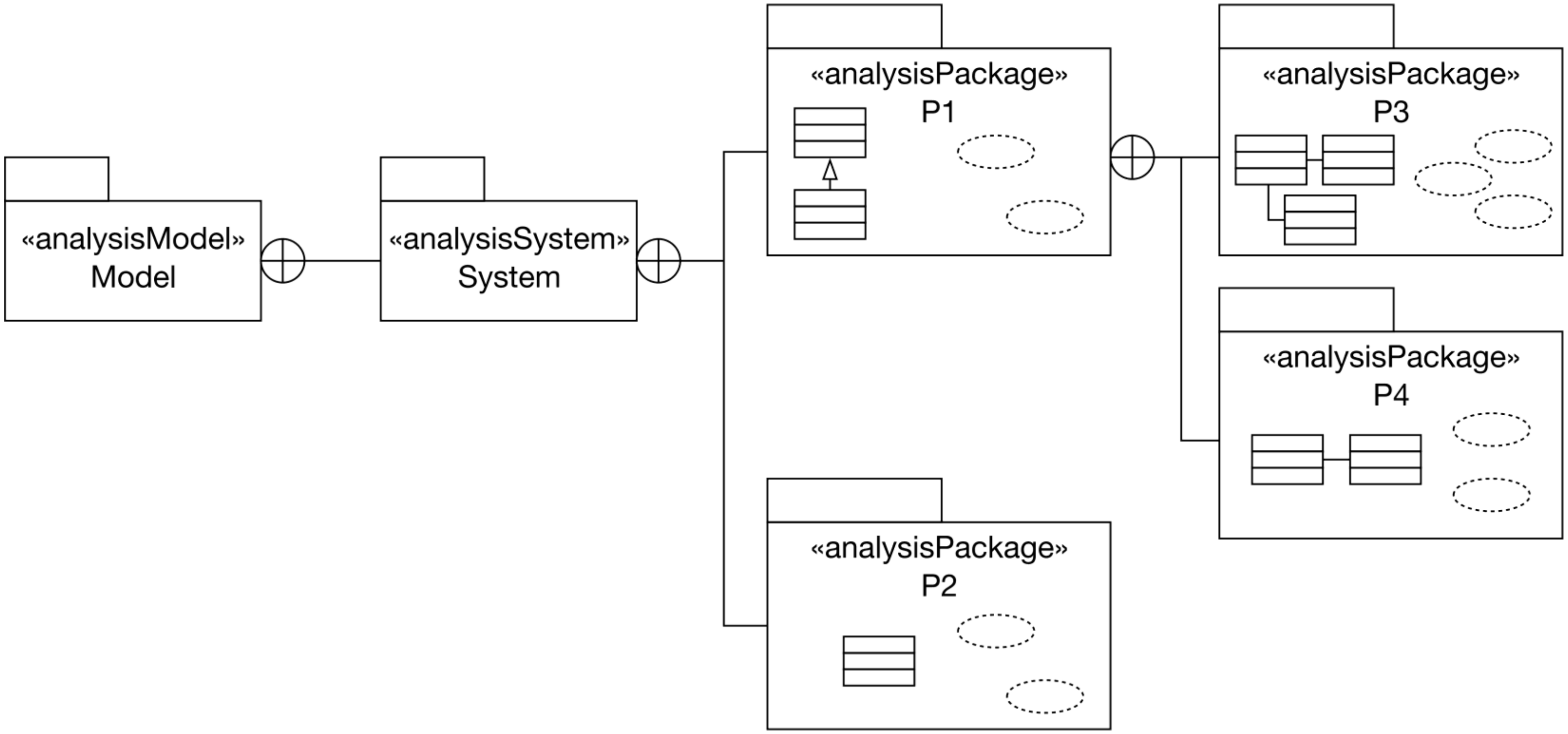


condition

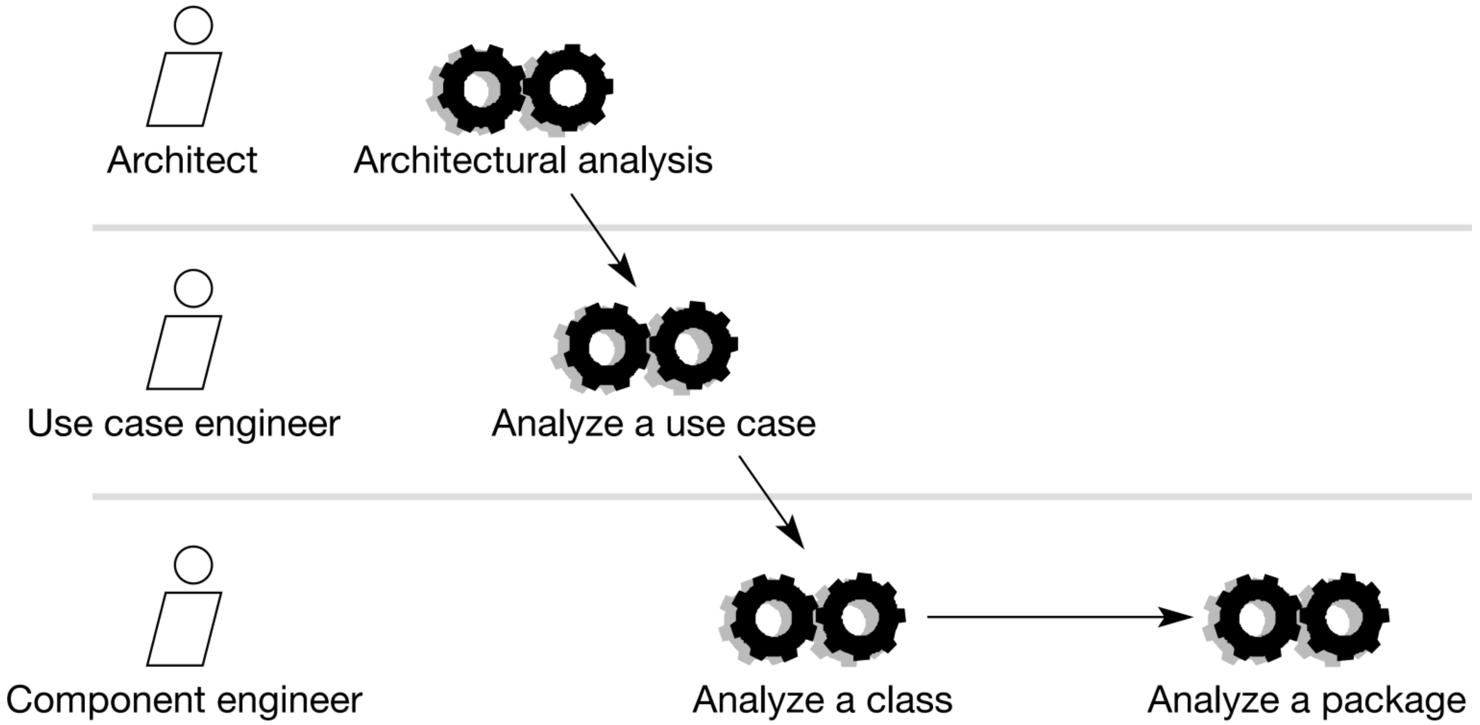
2. The Analysis workflow

- ✓ The aim is to produce an analysis model on *what* the system needs to do, leaving details on *how* it will do it to the design workflow

- ✓ Key artefacts produced: *analysis classes* (model key concepts in the business domain) and *use case realizations* (illustrate how instances of the analysis classes can interact to realize system behavior specified by a use case).



- ✓ Analysis workflow

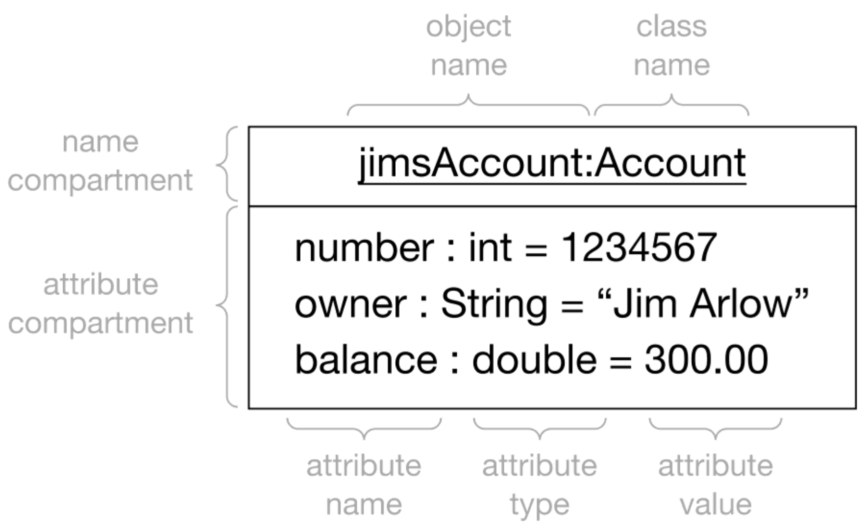


- ✓ Only classes part of the *vocabulary of the problem domain* (no design classes such as communications of database access classes, unless the problem is about that)

- ✓ Distinguish between the *problem domain* (business requirements) and the *solution domain* (design considerations)

- ✓ Is the model useful to all the stakeholders (subjects with a business interest)

✓ UML object notation:



object name

class name

name compartment

jimsAccount:Account

attribute compartment

number : int = 1234567

owner : String = “Jim Arlow”

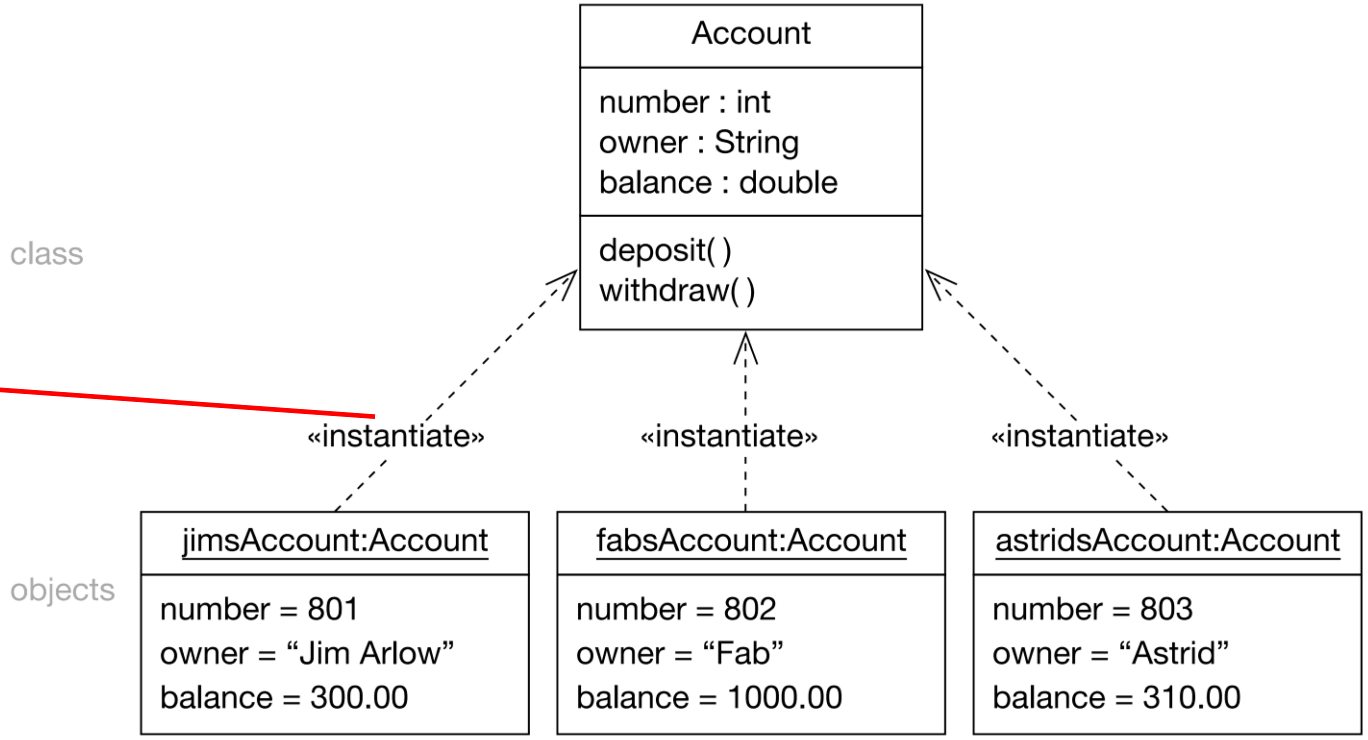
balance : double = 300.00

attribute name

attribute type

attribute value

✓ UML class notation:



class

Account

number : int
owner : String
balance : double

deposit( )
withdraw( )

«instantiate»

«instantiate»

«instantiate»

objects

jimsAccount:Account

number = 801
owner = “Jim Arlow”
balance = 300.00

fabsAccount:Account

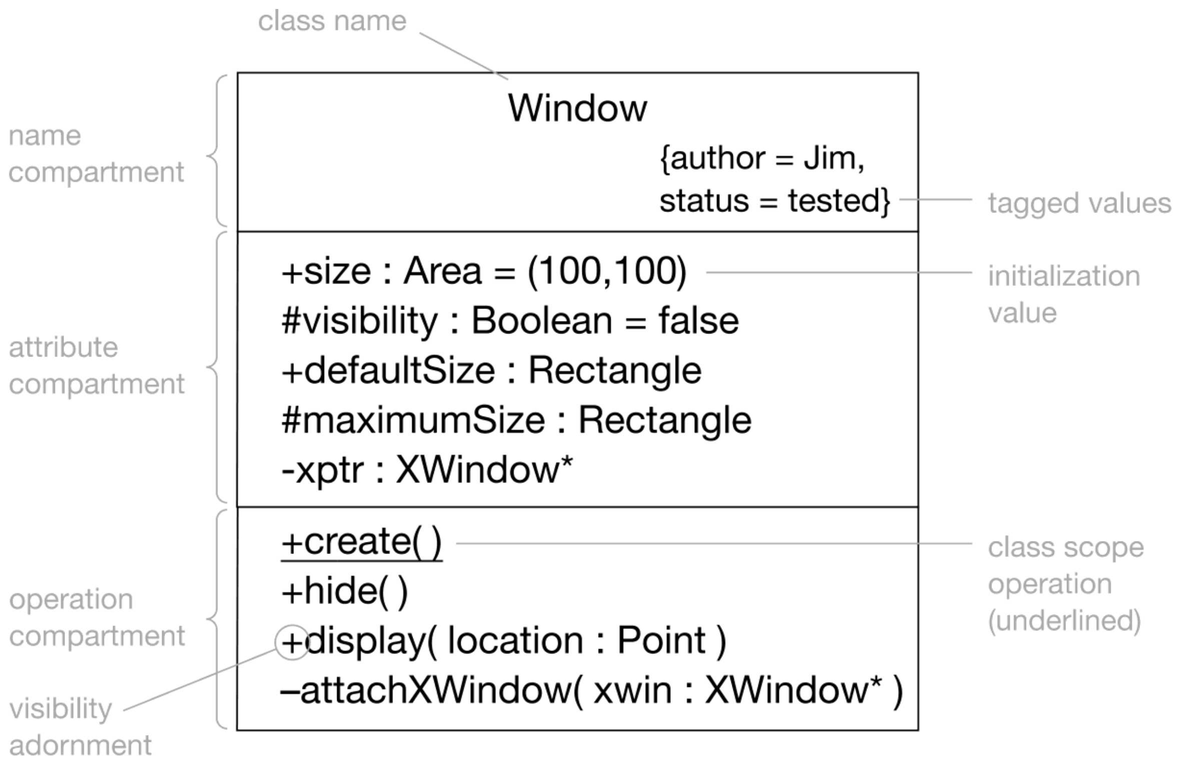
number = 802
owner = “Fab”
balance = 1000.00

astridsAccount:Account

number = 803
owner = “Astrid”
balance = 310.00

dependency relationship: a change to the class affects the object

- ✓ UML class notation:



class name

name compartment

Window

{author = Jim,
status = tested}

tagged values

attribute compartment

+size : Area = (100,100)
#visibility : Boolean = false
+defaultSize : Rectangle
#maximumSize : Rectangle
-xptr : XWindow*

initialization value

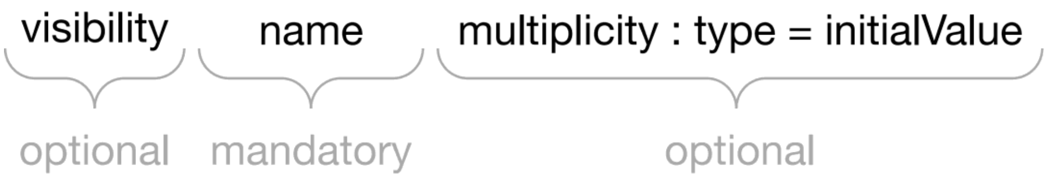
operation compartment

+create( )
+hide( )
+display(location : Point)
–attachXWindow(xwin : XWindow*)

class scope operation (underlined)

visibility adornment

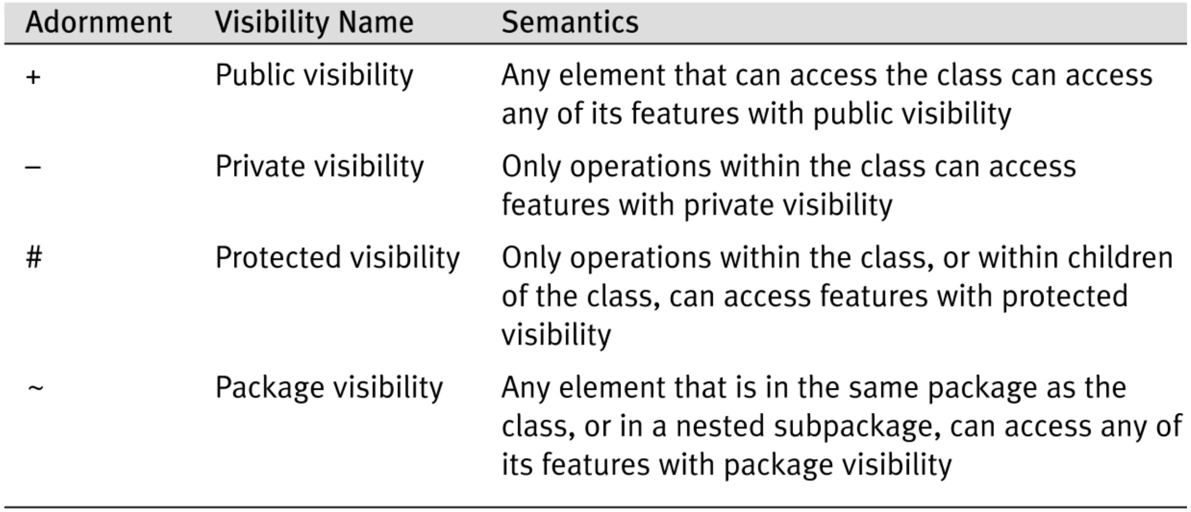
- ✓ Class name is CamelCase (no spaces or special symbols because they are used in languages)
- ✓ Avoid abbreviations of class name
- ✓ Attribute compartment:



visibility name multiplicity : type = initialValue

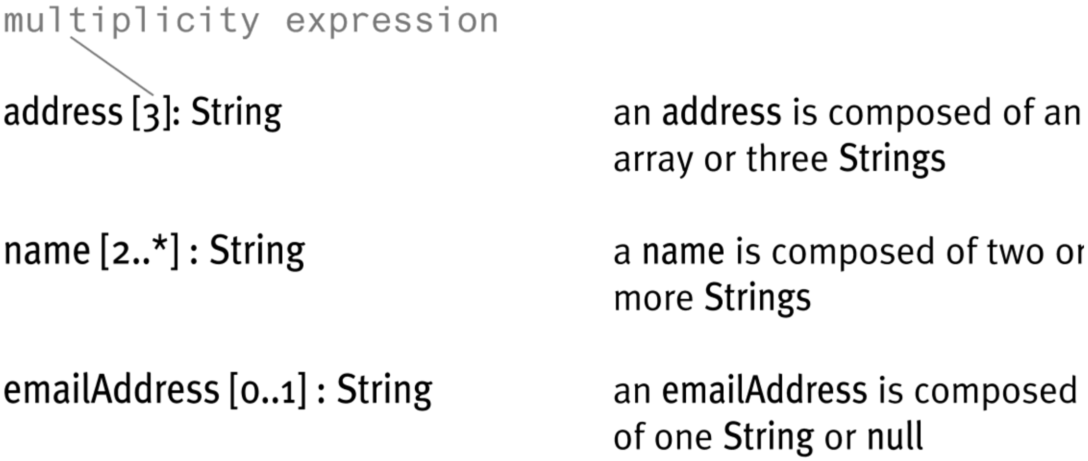
optional mandatory optional

- ✓ Visibility adornment:

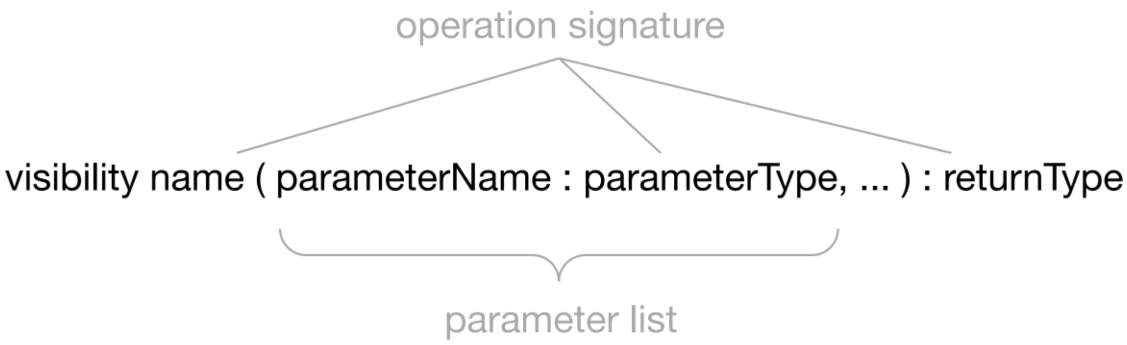


Adornment	Visibility Name	Semantics
+	Public visibility	Any element that can access the class can access any of its features with public visibility
–	Private visibility	Only operations within the class can access features with private visibility
#	Protected visibility	Only operations within the class, or within children of the class, can access features with protected visibility
~	Package visibility	Any element that is in the same package as the class, or in a nested subpackage, can access any of its features with package visibility

- ✓ Initial values and visibility are not used in the analysis model.
- ✓ Multiplicity (number of things) is more used in design, sometimes in analysis:

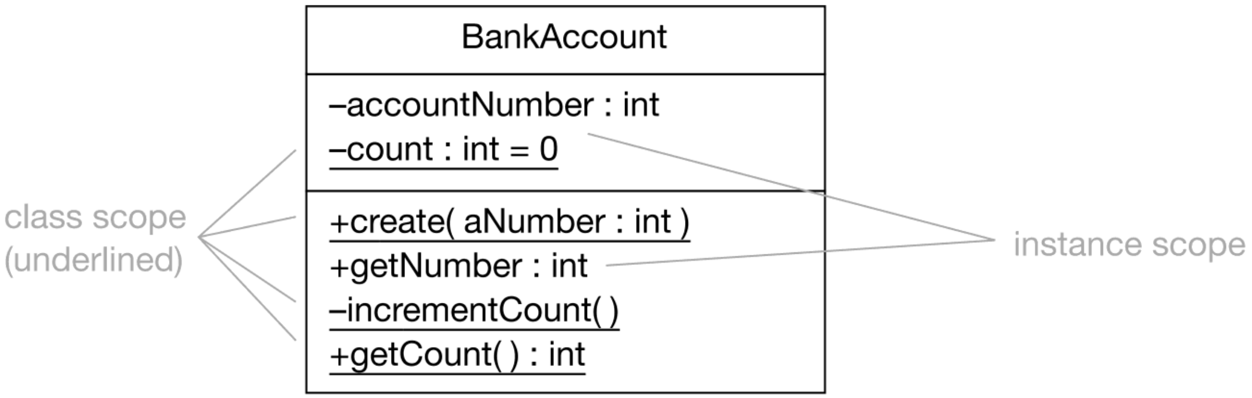


address [3]: String	an address is composed of an array or three Strings
name [2..*] : String	a name is composed of two or more Strings
emailAddress [0..1] : String	an emailAddress is composed of one String or null


operation signature
visibility name (parameterName : parameterType, ...) : returnType
parameter list

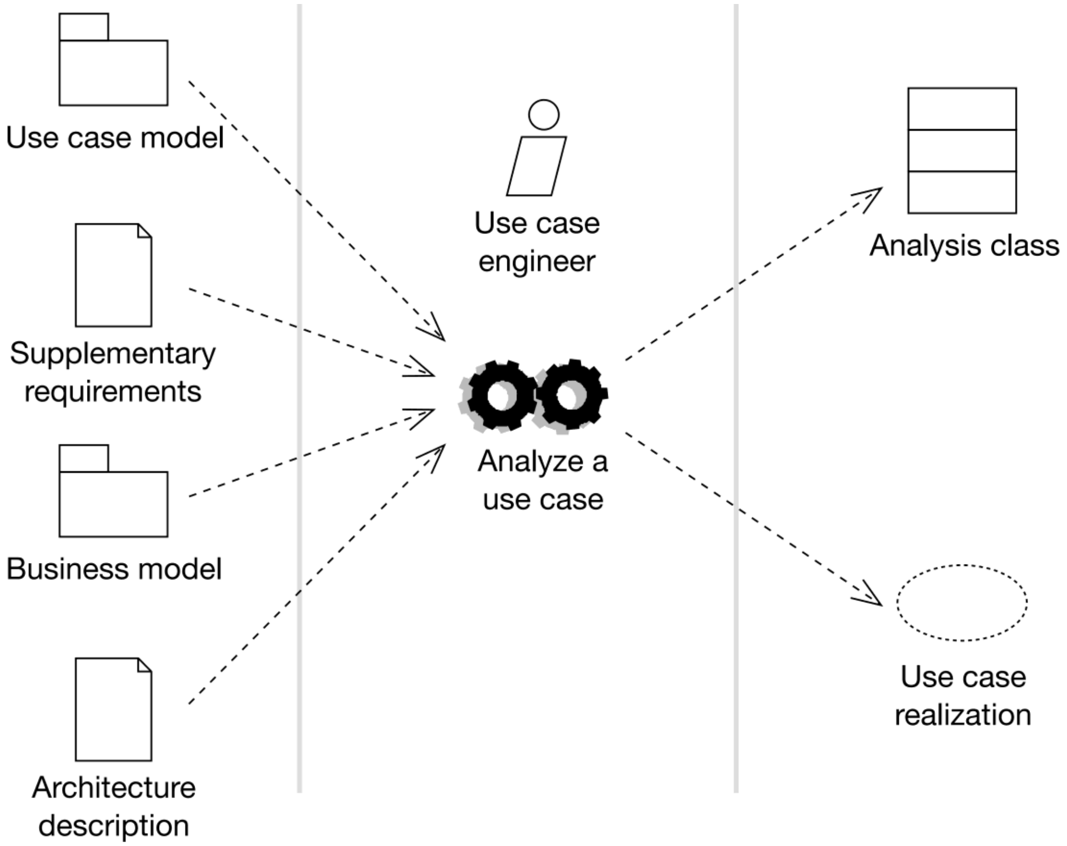
- ✓ Operation compartment:

- ✓ Instance and class scope (one version shared by all objects):



- ✓ Activity “analyze a use case”: creating analysis classes and use case realizations

- ✓ *Analysis class* is in the problem domain (in which the need for the system arises)



Use case model

Supplementary requirements

Business model

Architecture description

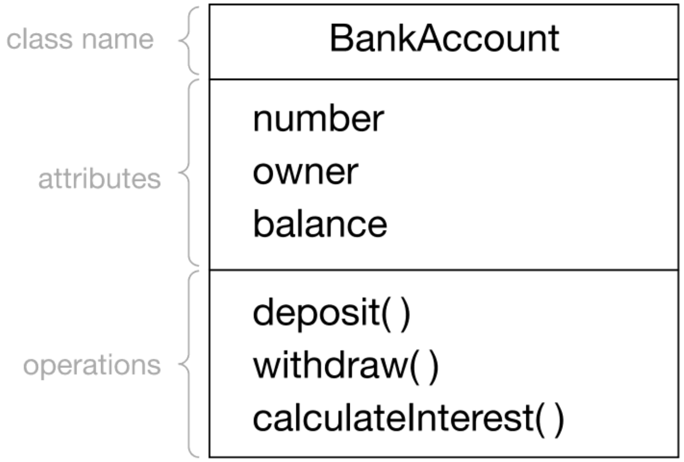
Use case engineer

Analyze a use case

Analysis class

Use case realization

- ✓ Anatomy of an analysis class:



class name

attributes

operations

BankAccount

number
owner
balance

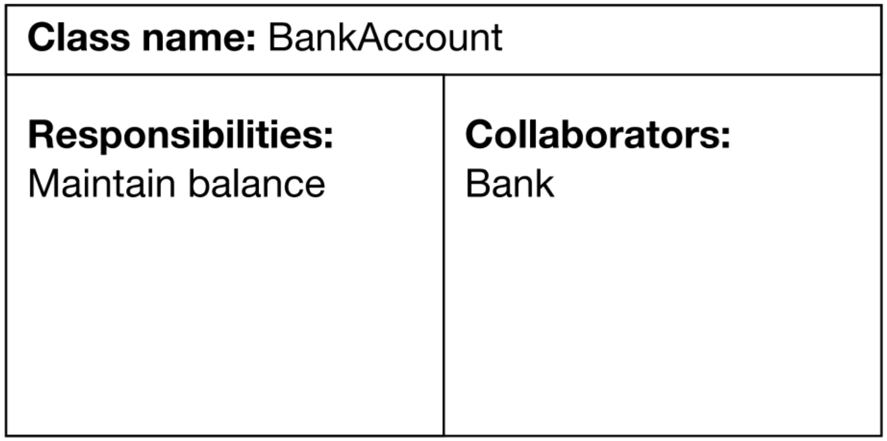
deposit( )
withdraw( )
calculateInterest( )

- its name reflects its intent;
- it is a crisp abstraction that models one specific element of the problem domain;
- it maps on to a clearly identifiable feature of the problem domain;
- it has a small, well-defined set of responsibilities;
- it has high cohesion (cohesive set of responsibilities towards the same goal);
- it has low coupling to other classes (number of relationships).

✓ Beware of large classes, functoids, omnipotent classes, deep inheritance

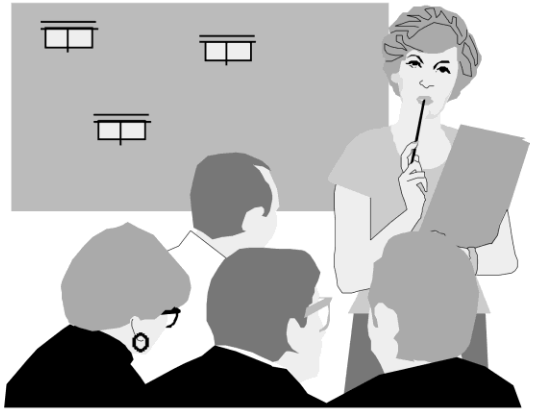
✓ How to find analysis classes: noun and noun phrases indicate candidate classes or attributes, whereas verb and verb phrases indicate candidate responsibilities.

✓ CRC (Clas Responsibilities Collaboration):

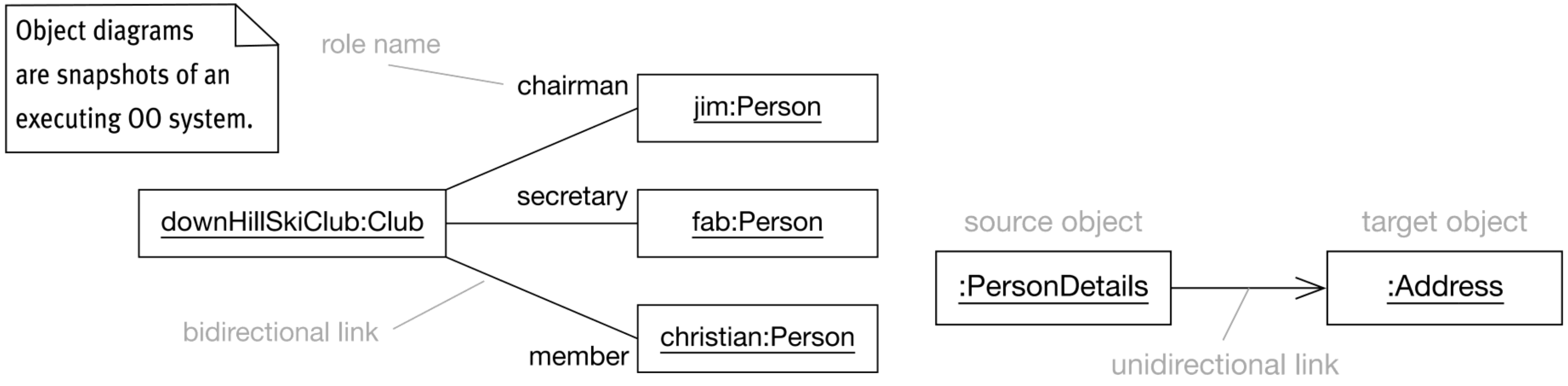


Class name: BankAccount

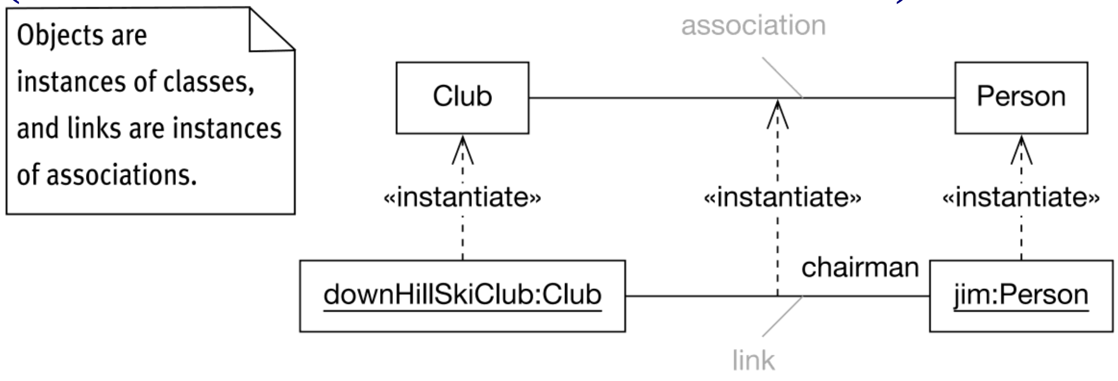
Responsibilities:	**Collaborators:**
Maintain balance	Bank



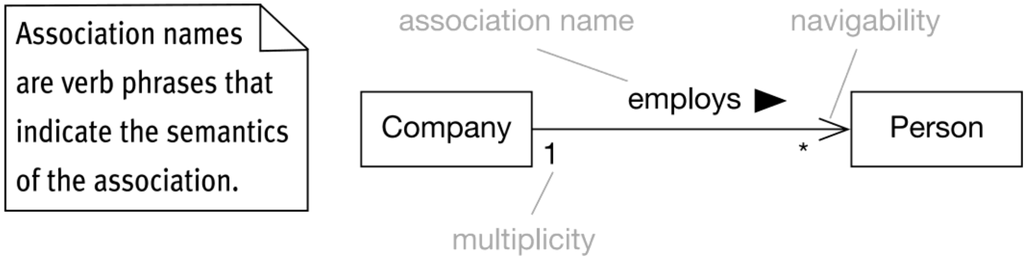
- ✓ *Link* in object diagram: it allows messages to be sent from one object to the other (pointer, references, etc.)



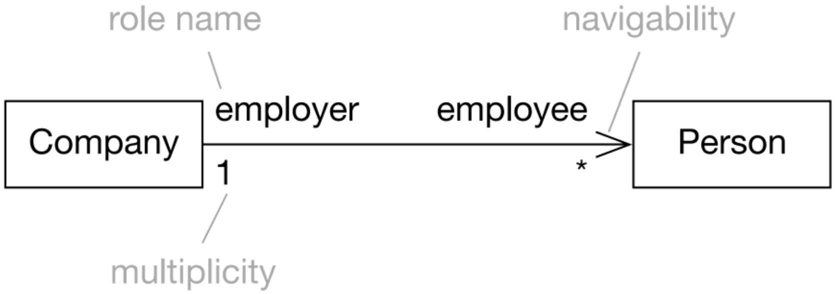
- ✓ *Association* in class diagram: relationship between classes (a link is an instantiation of an association)



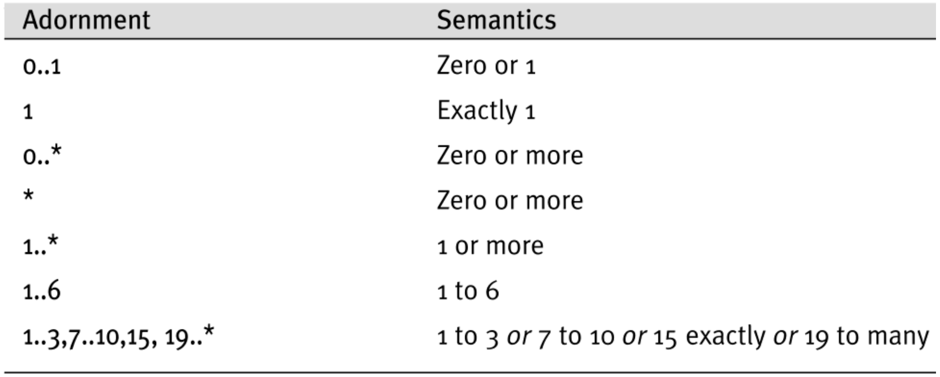
- ✓ “A *Company* employs many *Persons* (a black triangle denotes the reading direction), or “Each *Person* works for one *Company*” at any point in time.
  Over time a Person object might be employed by a sequence of Company objects.


Association names are verb phrases that indicate the semantics of the association.
association name
navigability
employs ▶
Company
Person
1
*
multiplicity

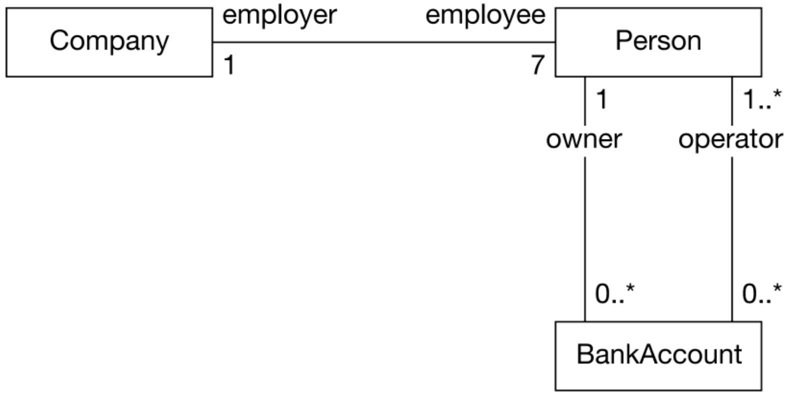
- ✓ Associations can have roles instead of association name:



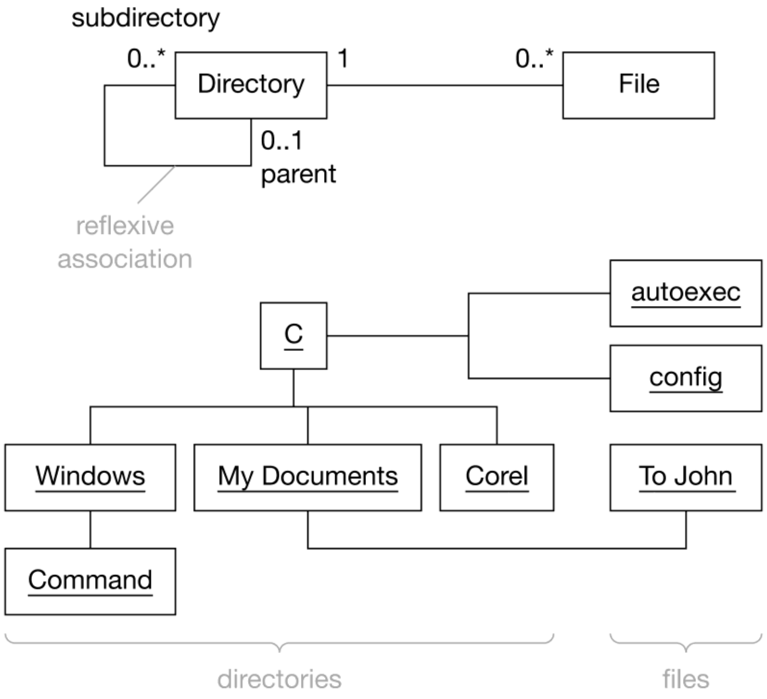
✓ Multiplicity (there is not “default” multiplicity if it is not explicitly stated):

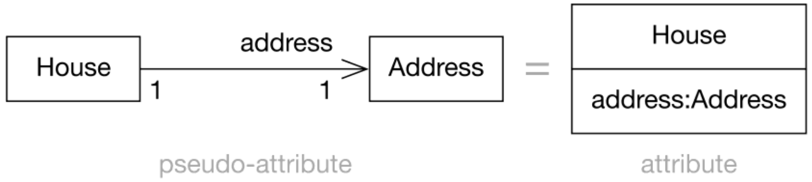


Adornment	Semantics
0..1	Zero or 1
1	Exactly 1
0..*	Zero or more
*	Zero or more
1..*	1 or more
1..6	1 to 6
1..3,7..10,15, 19..*	1 to 3 *or* 7 to 10 *or* 15 exactly *or* 19 to many



✓ Reflexive associations

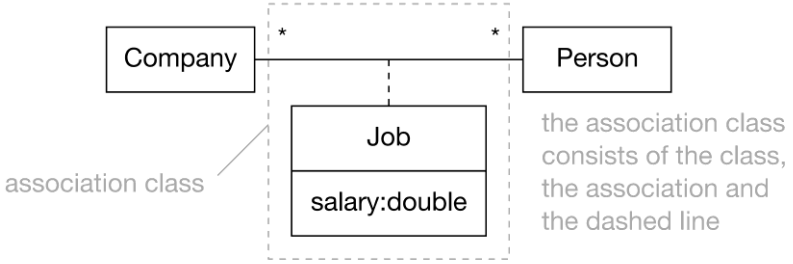




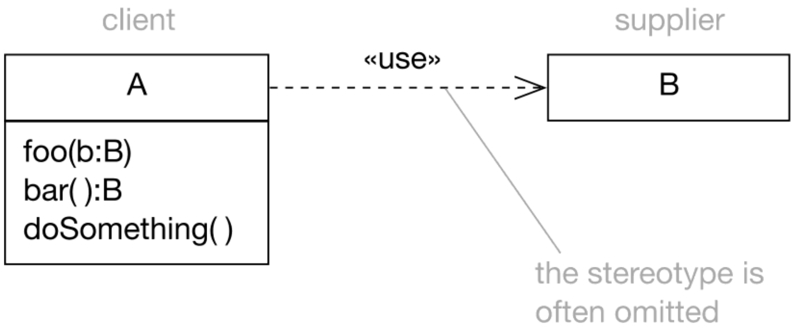
```
public class House
{   private Address address;
}
```

- ✓ Implementation of association as an attribute

- ✓ Association class (association that is also a class)

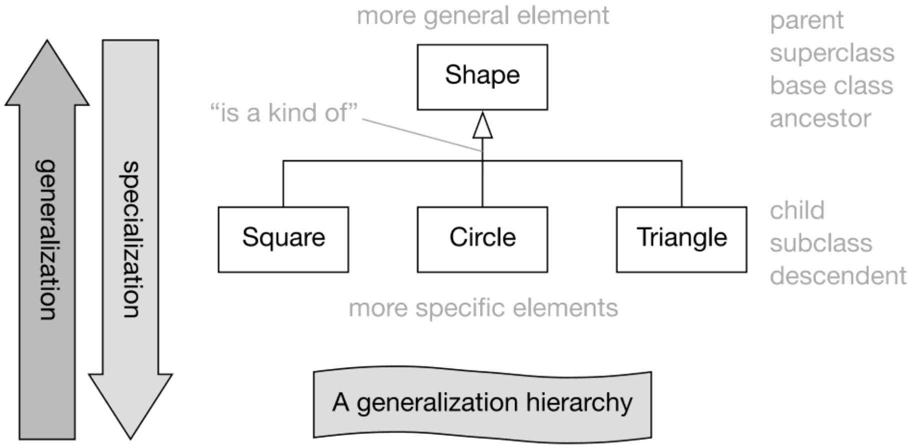


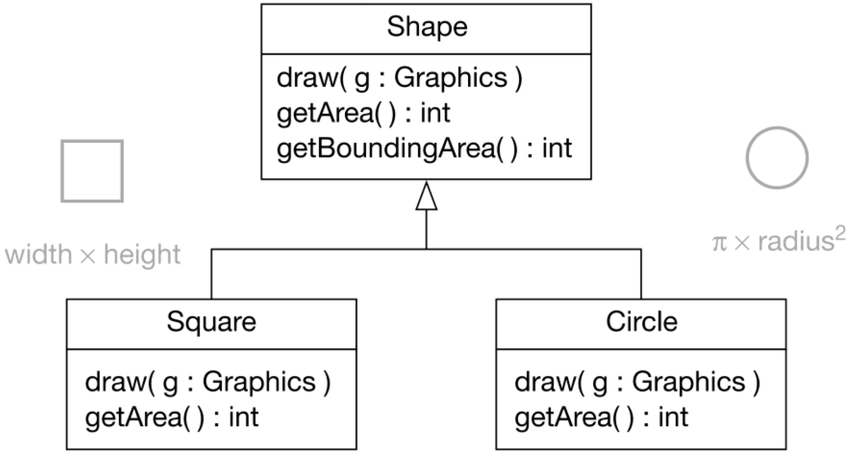
- ✓ Dependency (between classes, packages, object and classes)

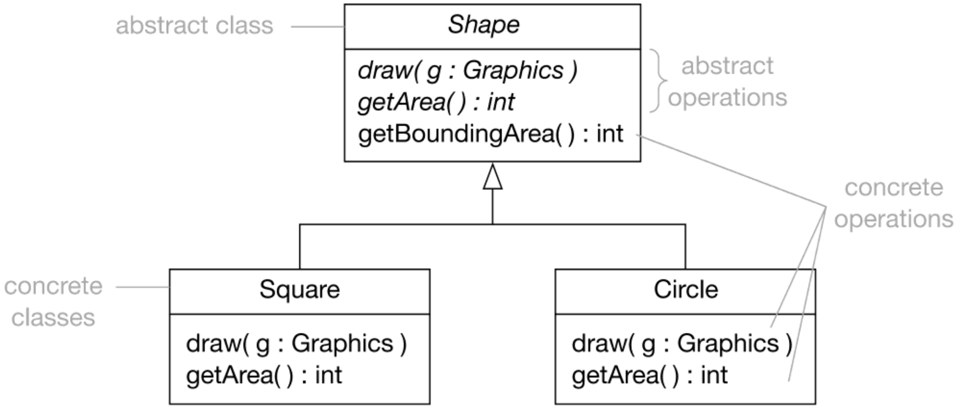


  - ✓ The «use» dependency:
    An operation of class A needs a parameter, returns a value, uses an object of class B somewhere in its implementation, but not as an attribute

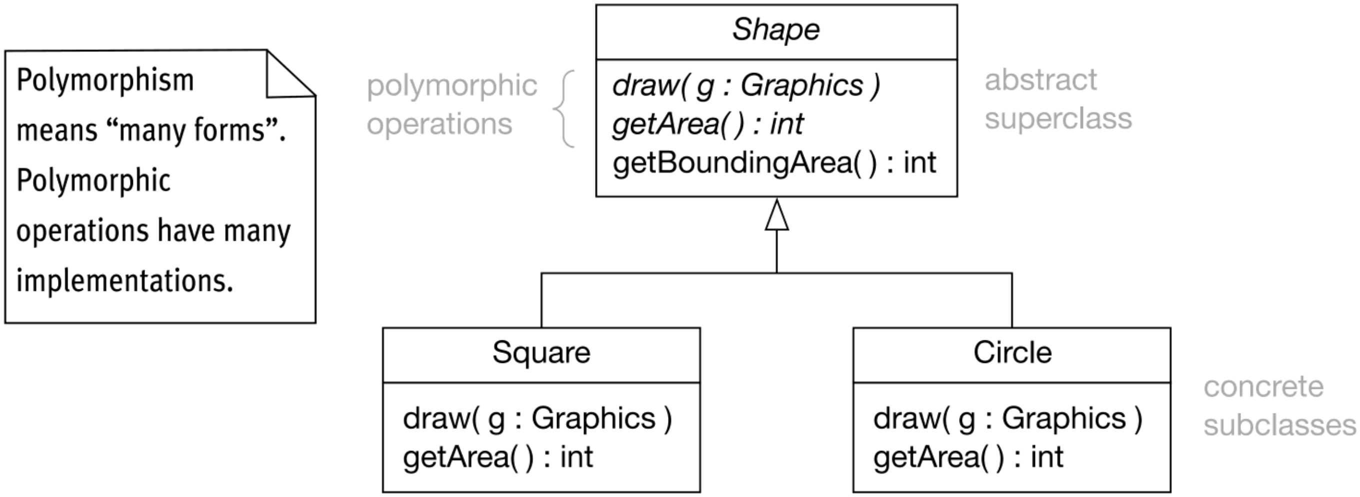
- ✓ The «call» dependency: an operation of class A invokes an operation of class B
- ✓ The «parameter» dependency: in class B, a parameter or returned value of class A
- ✓ The «send» dependency: a class A transfers data to a class B
- ✓ The «intantiate» dependency: an instance of class A
- ✓ The «access» dependency: a package P accesses the public content of package Q (Packages are used in UML to group things)
- ✓ The «import» dependency: the namespace of a package P is merged to the namespace of package Q (you do not need a qualified element name)
- ✓ Generalization: specialized (or extended) classes inherit attributes, operations, relationships, constraints. Overriding of operations (same signature)







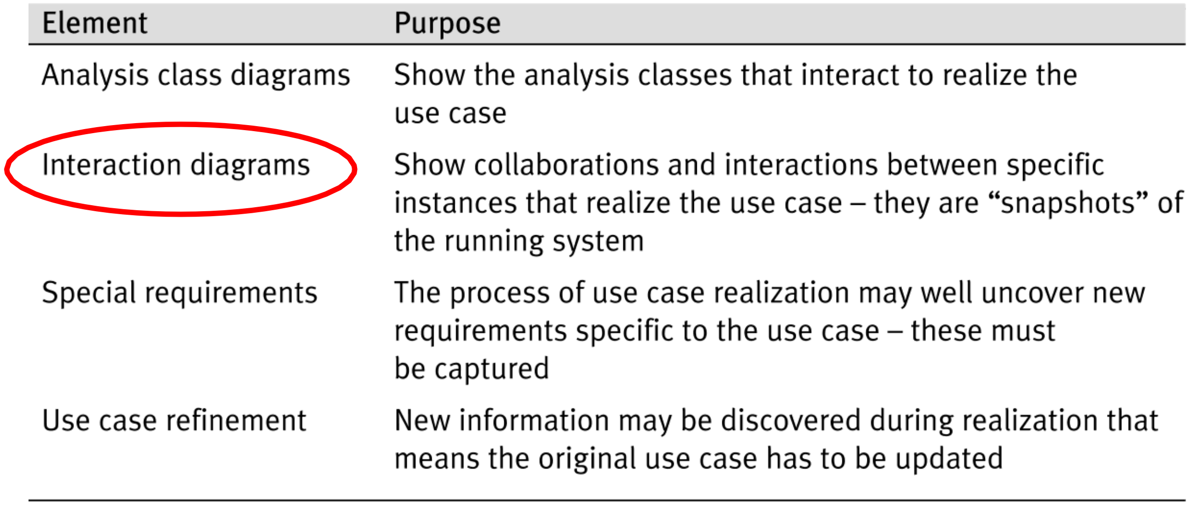
- ✓ Abstract class cannot be instantiated.


Polymorphism means “many forms”. Polymorphic operations have many implementations.

- ✓ Polymorphism:
  there are two implementations of the Shape class, i.e., its operations have many forms (polymorphic) depending on the class of its instance (Square or Circle)

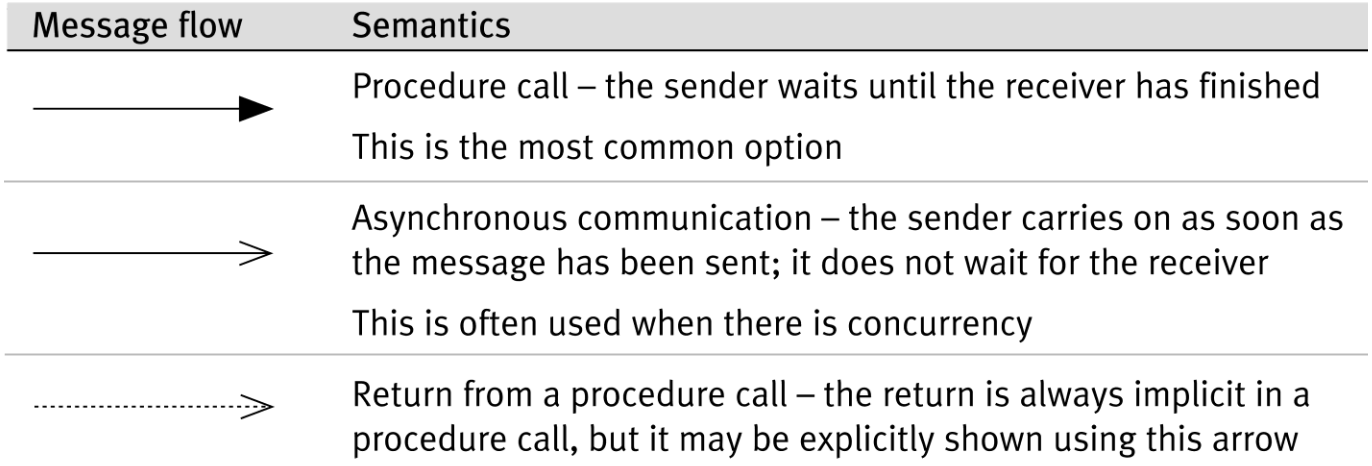
- ✓ Overriding concrete operations is considered a bad style.

- ✓ *Dynamic view*: use case realizations show how instances of the analysis classes interact to realize the functionality of the system, via the following elements:



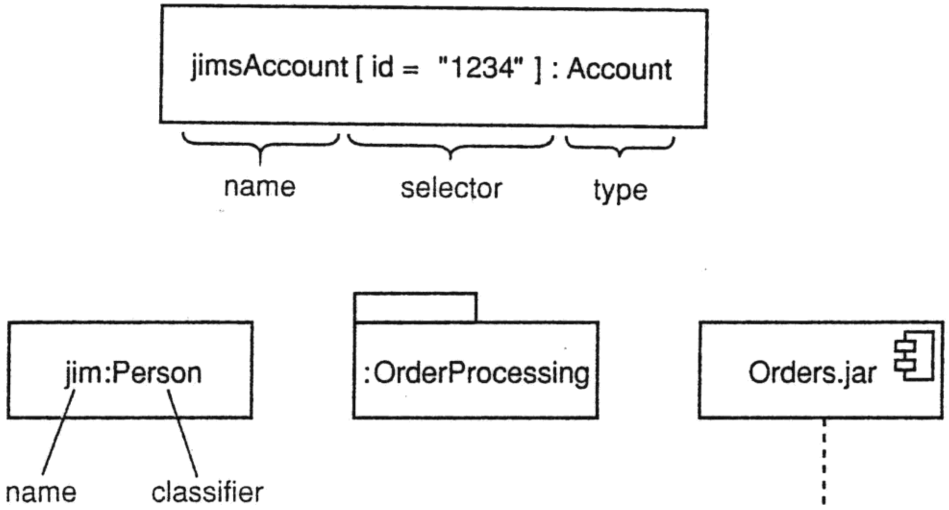
Element	Purpose
Analysis class diagrams	Show the analysis classes that interact to realize the use case
Interaction diagrams	Show collaborations and interactions between specific instances that realize the use case – they are “snapshots” of the running system
Special requirements	The process of use case realization may well uncover new requirements specific to the use case – these must be captured
Use case refinement	New information may be discovered during realization that means the original use case has to be updated

- ✓ Types of interaction diagrams: *communication diagram* and *sequence diagram* (dynamic interaction between instances in terms of *messages*).



Message flow	Semantics
——▶	Procedure call – the sender waits until the receiver has finished This is the most common option
——>	Asynchronous communication – the sender carries on as soon as the message has been sent; it does not wait for the receiver This is often used when there is concurrency
·········>	Return from a procedure call – the return is always implicit in a procedure call, but it may be explicitly shown using this arrow

- *Lifeline*: a participant in an interaction, an instance of a specific classifier (a classifier is a type of thing, such as actor, class, use case; an instance is a concrete example of such thing such as a specific actor, class, use case).

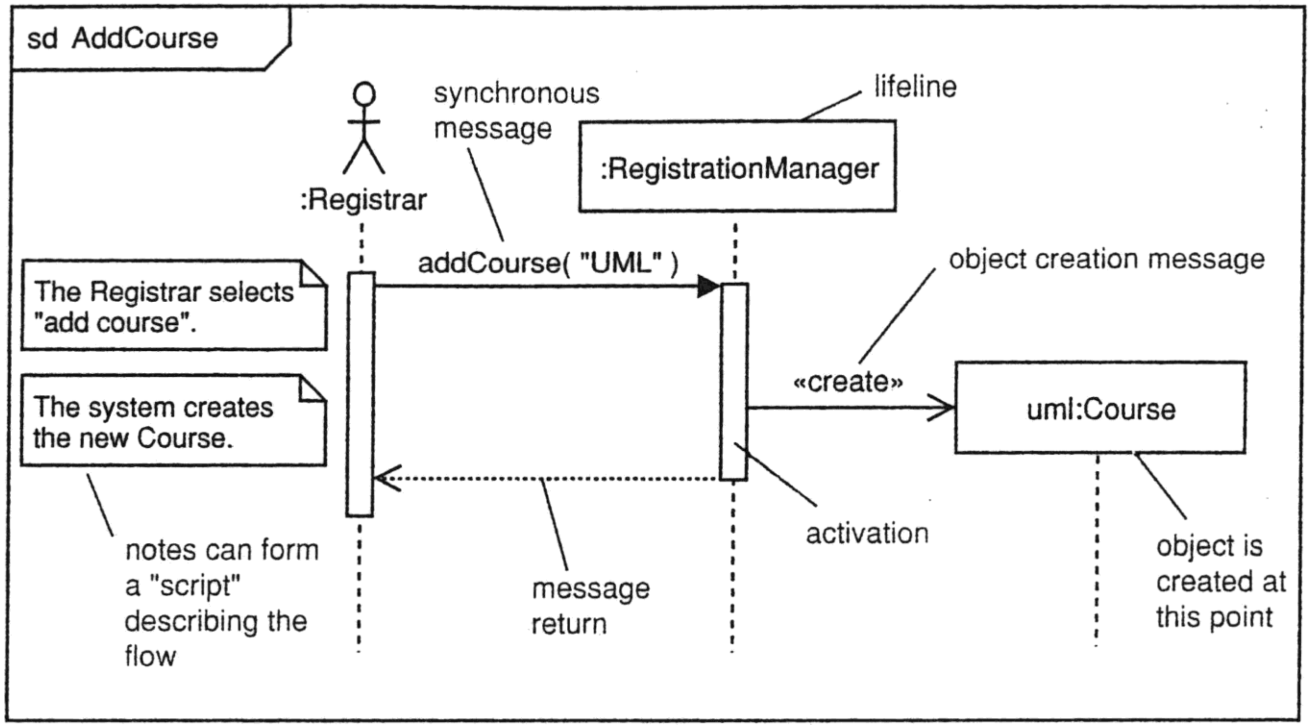


- *Selector*: a Boolean condition to select a single instance

- Interaction diagrams are not verbatim transcriptions of a use case, they are illustrations of how the use case behavior is realized by analysis classes

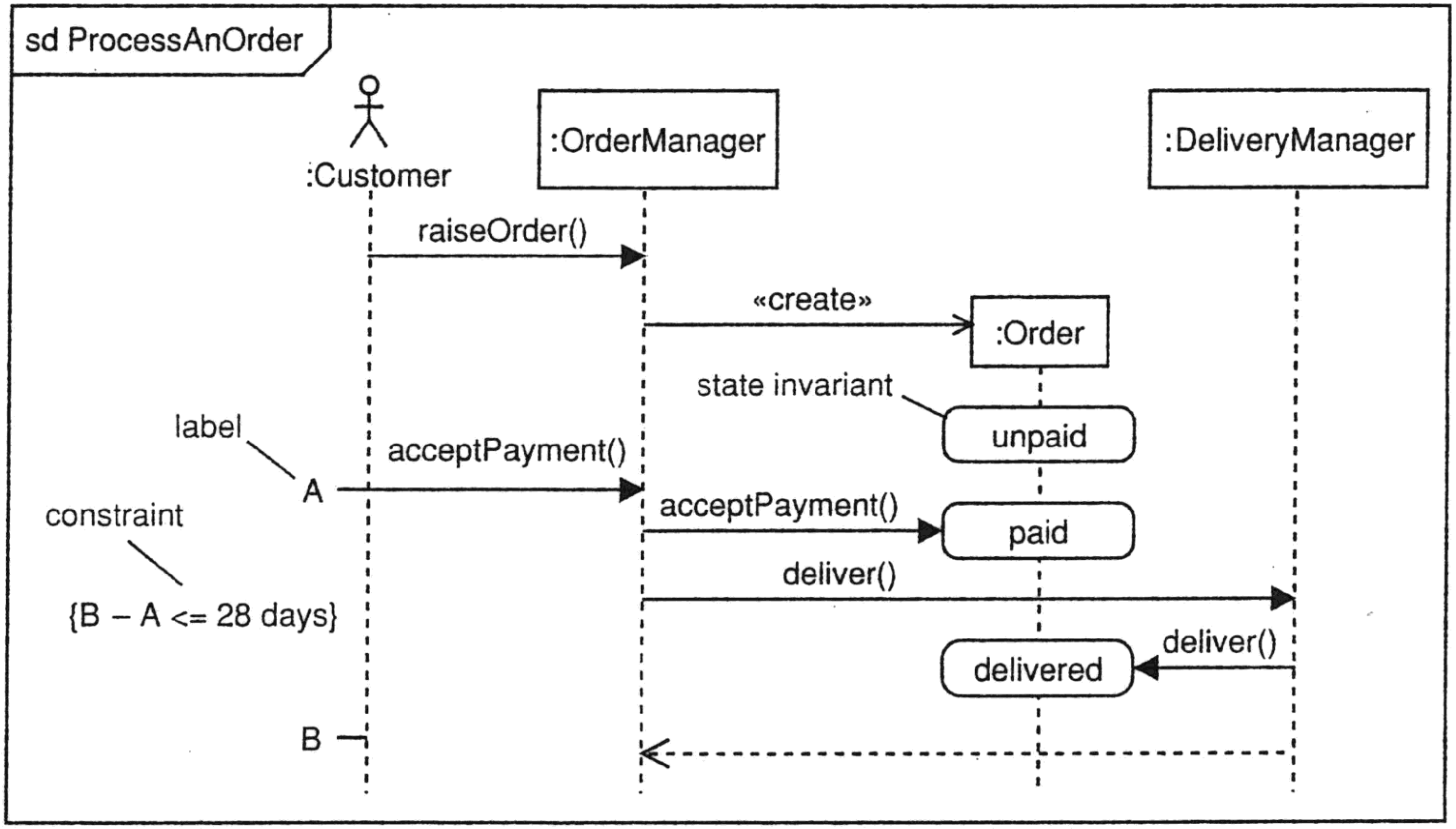
- Use case and sequence diagram

Use case: AddCourse
ID: UC8
Actors: Registrar
Preconditions: The Registrar has logged on to the system.
Flow of events: 1. The Registrar selects “add course”. 2. The system accepts the name of the new course. 3. The system creates the new course.
Postconditions: A new course has been added to the system.

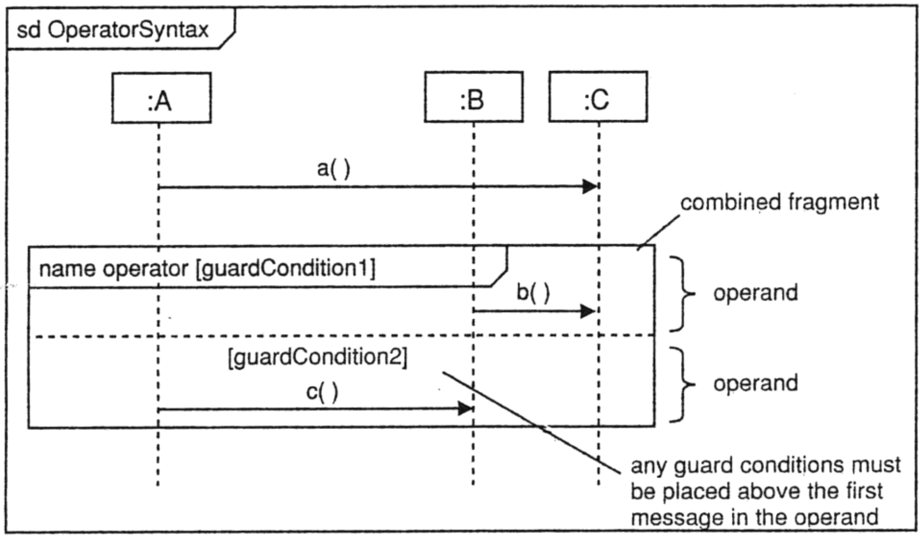


✓ State invariants and constraints: a classifier can have a state machine describing the life cycle of its instances in terms of states and events causing transition between states

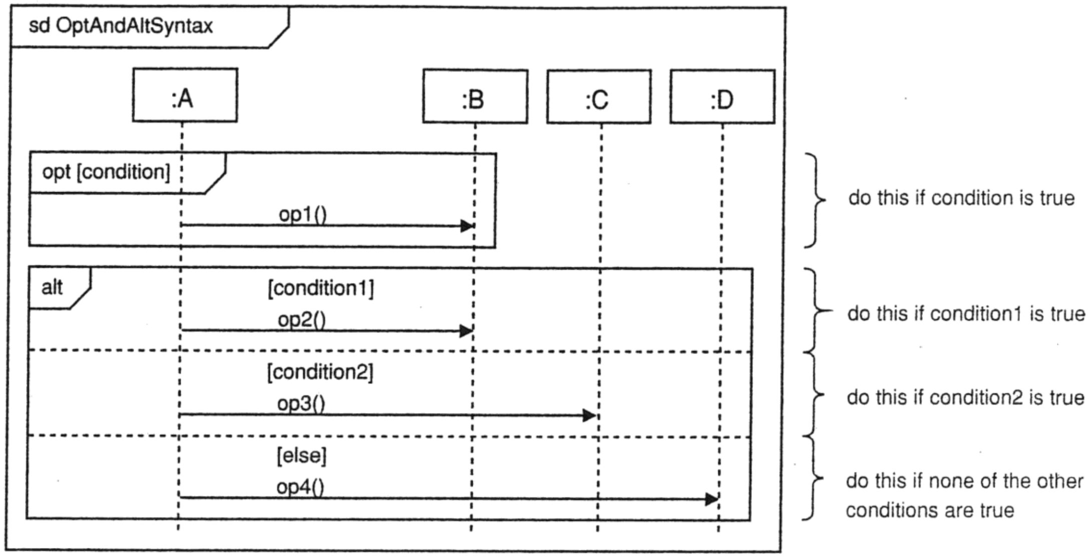
✓ if a message causes a state change, lifelines can show the state of the instances. Example of constraints: the order shall be delivered no more than 28 days after payment has been received.

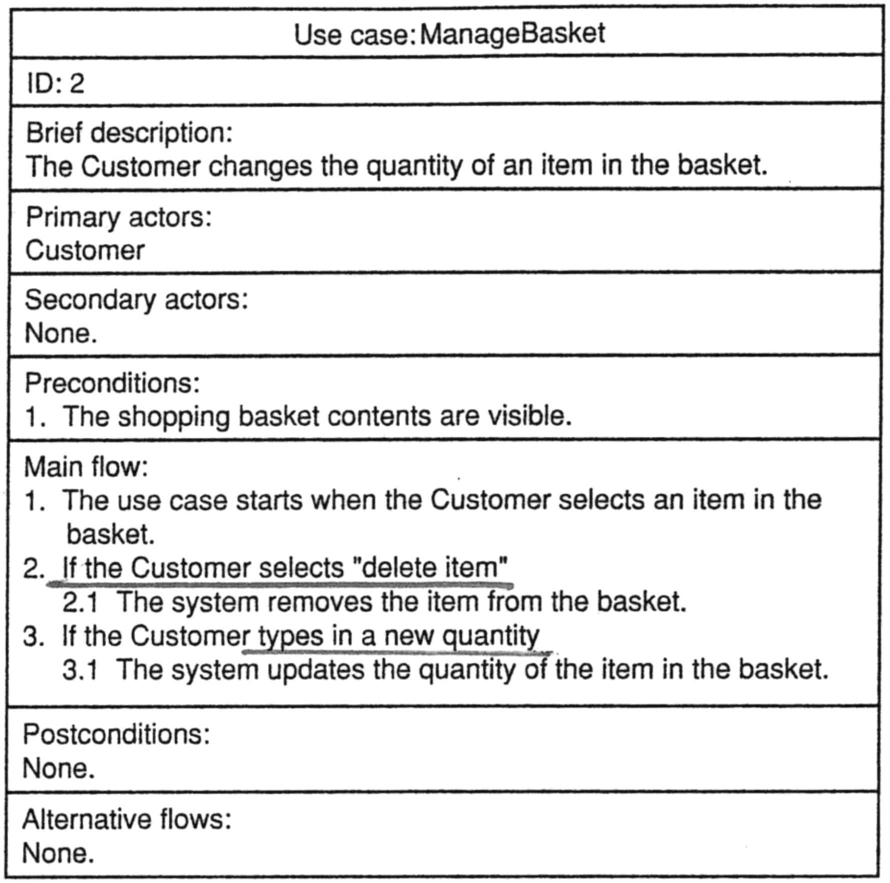


- ✓ Combined fragment and operators: combined fragments are areas of the sequence diagram; the operator determines *how* its operands are executed, whereas the guard condition determines *whether* their operand execute.

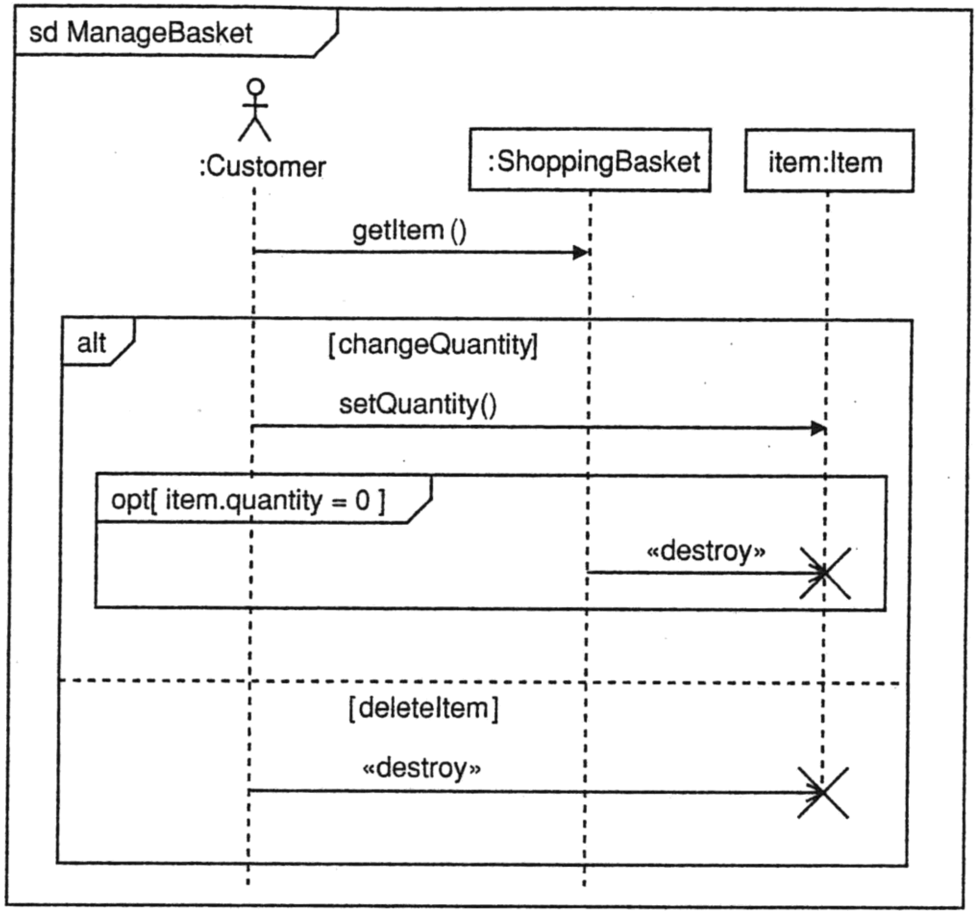

sd OperatorSyntax
:A
:B
:C
a( )
combined fragment
name operator [guardCondition1]
b( )
operand
[guardCondition2]
c( )
operand
any guard conditions must be placed above the first message in the operand

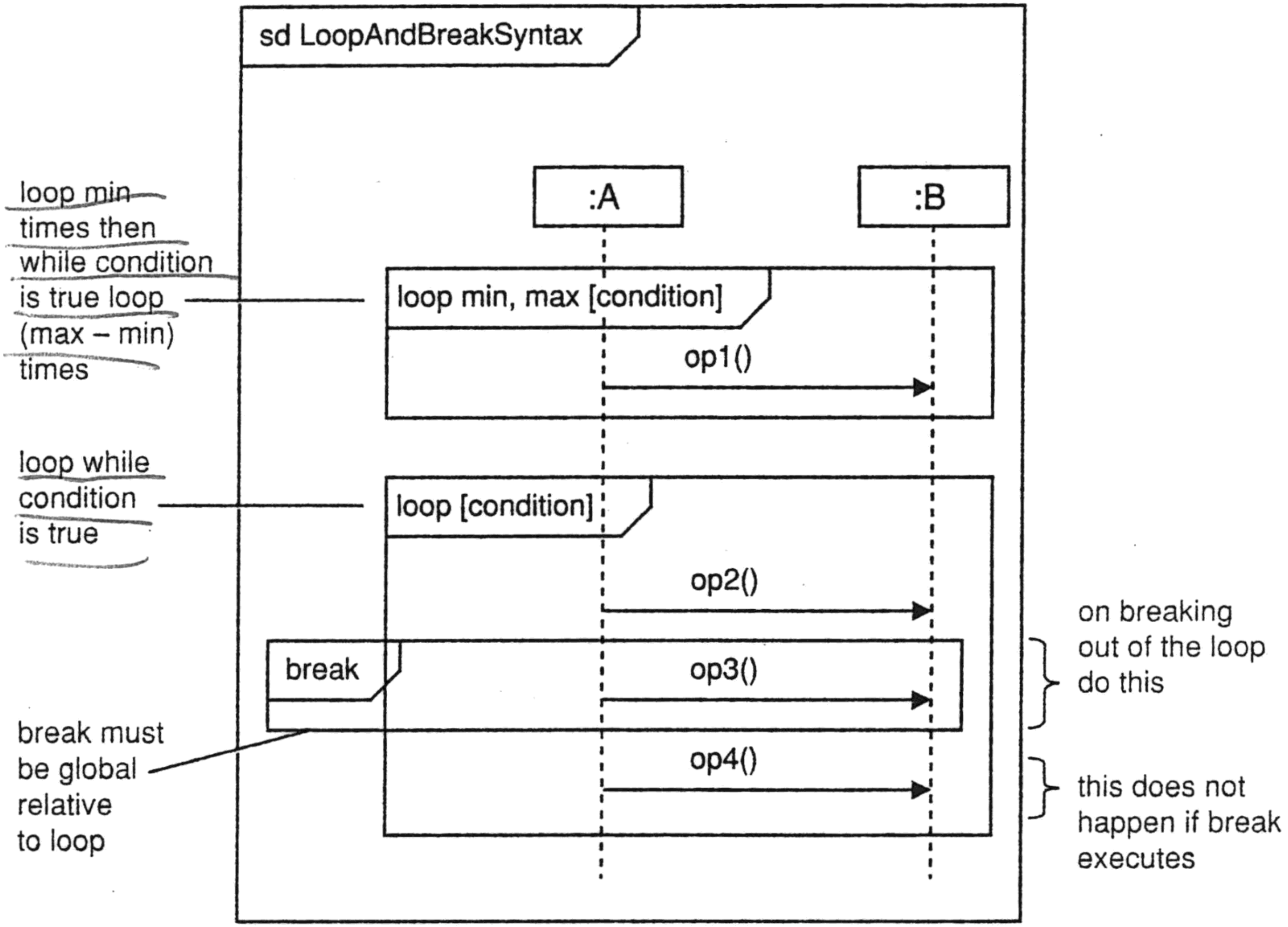
```
if (condition1) then
  operand 1
else if (condition2) then
  operand 2
...
else if (conditionN) then
  operand N
else
  operand M
```


sd OptAndAltSyntax
:A
:B
:C
:D
opt [condition]
op1()
do this if condition is true
alt
[condition1]
op2()
do this if condition1 is true
[condition2]
op3()
do this if condition2 is true
[else]
op4()
do this if none of the other conditions are true



Use case: ManageBasket
ID: 2
Brief description: The Customer changes the quantity of an item in the basket.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The shopping basket contents are visible.
Main flow: 1. The use case starts when the Customer selects an item in the basket. 2. If the Customer selects "delete item" 2.1 The system removes the item from the basket. 3. If the Customer types in a new quantity 3.1 The system updates the quantity of the item in the basket.
Postconditions: None.
Alternative flows: None.





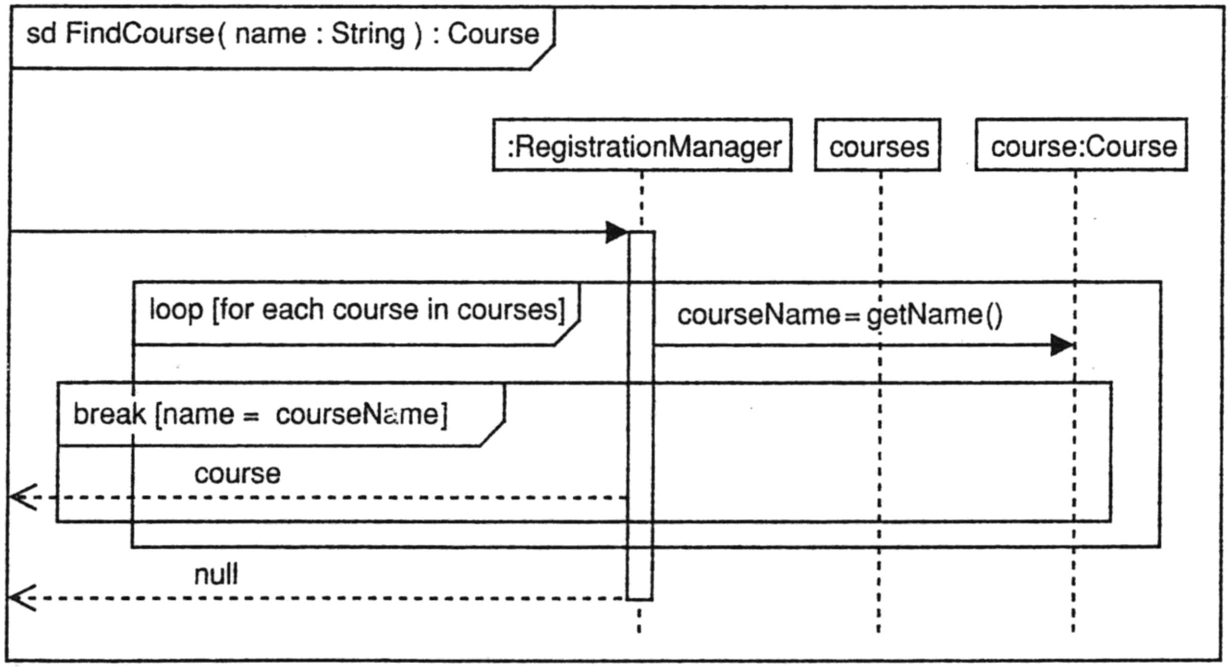
loop min times then while condition is true loop (max – min) times

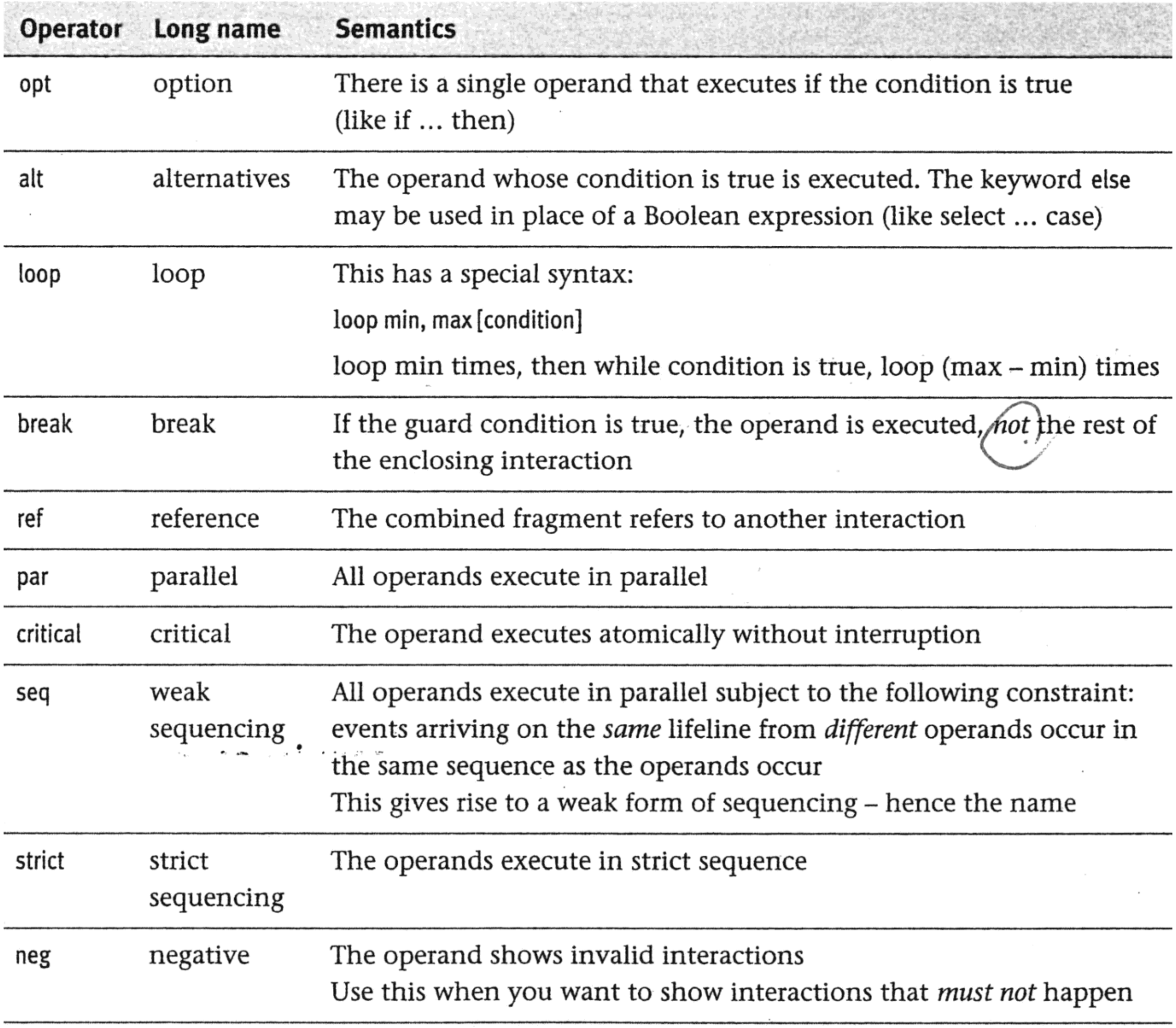
loop while condition is true

break must be global relative to loop

on breaking out of the loop do this

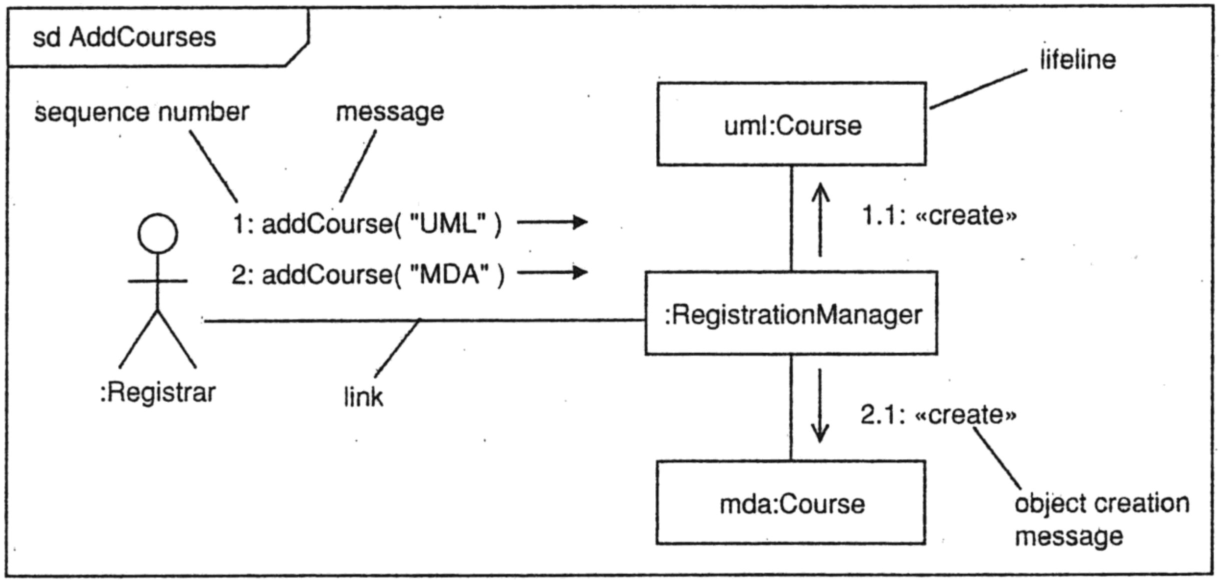
this does not happen if break executes

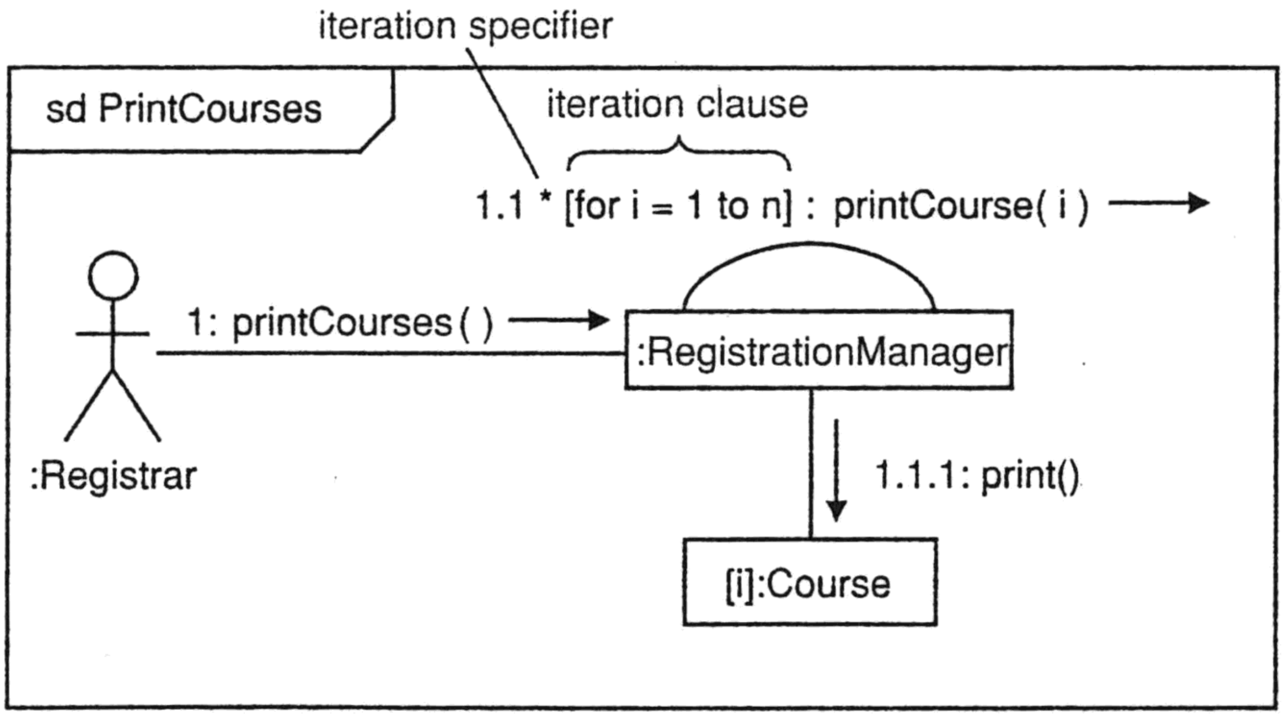

sd FindCourse(name : String) : Course
:RegistrationManager
courses
course:Course
loop [for each course in courses]
courseName = getName()
break [name = courseName]
course
null



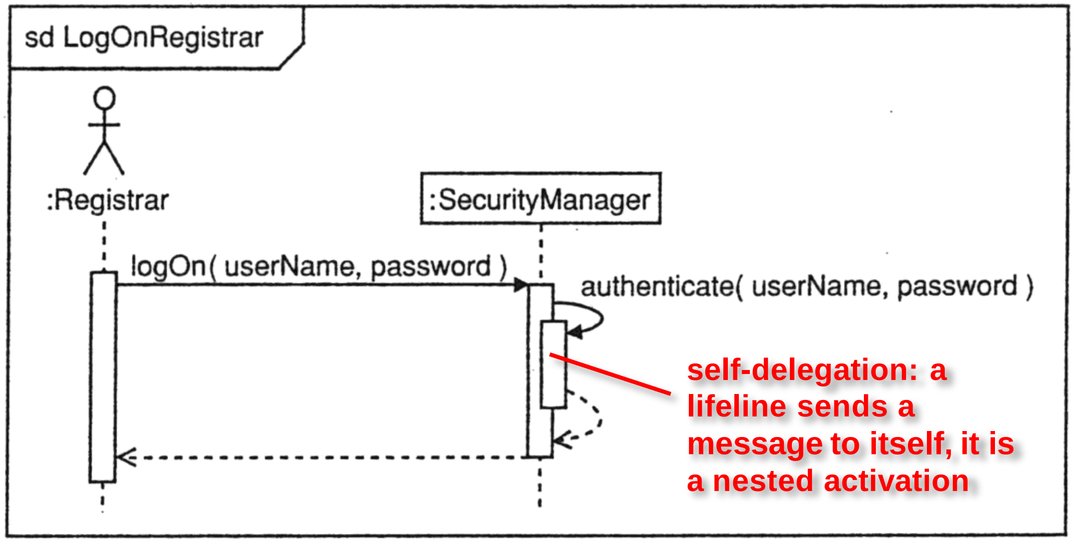
Operator	Long name	Semantics
opt	option	There is a single operand that executes if the condition is true (like if … then)
alt	alternatives	The operand whose condition is true is executed. The keyword else may be used in place of a Boolean expression (like select … case)
loop	loop	This has a special syntax: loop min, max [condition] loop min times, then while condition is true, loop (max – min) times
break	break	If the guard condition is true, the operand is executed, *not* the rest of the enclosing interaction
ref	reference	The combined fragment refers to another interaction
par	parallel	All operands execute in parallel
critical	critical	The operand executes atomically without interruption
seq	weak sequencing	All operands execute in parallel subject to the following constraint: events arriving on the *same* lifeline from *different* operands occur in the same sequence as the operands occur This gives rise to a weak form of sequencing – hence the name
strict	strict sequencing	The operands execute in strict sequence
neg	negative	The operand shows invalid interactions Use this when you want to show interactions that *must not* happen

- ✓ Communication diagram: it is similar to sequence diagram except that there are direct links between lifelines

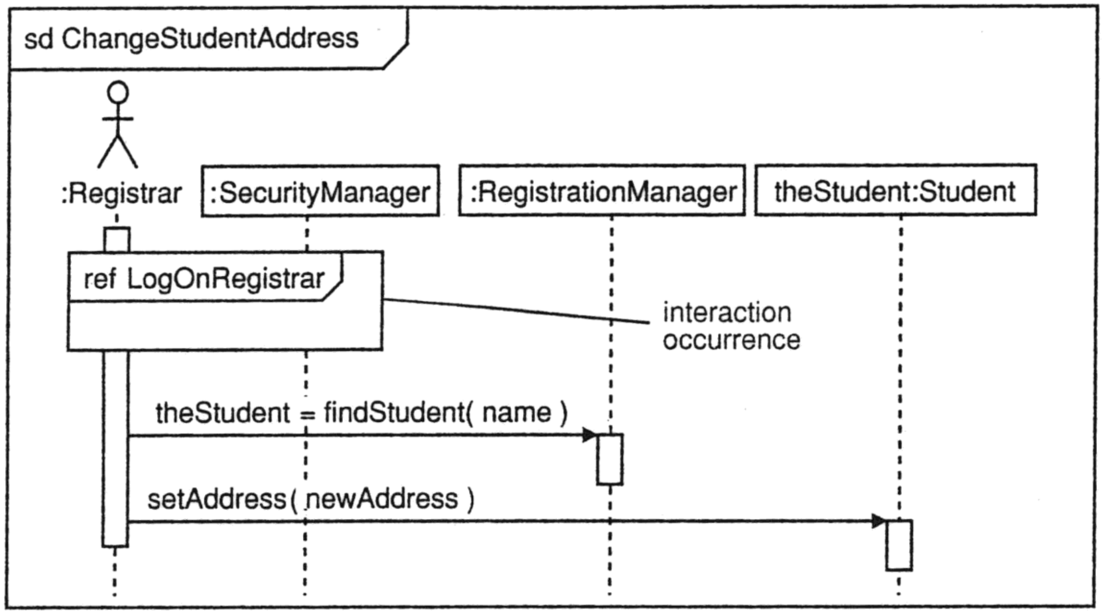

sd AddCourses
sequence number
message
lifeline
uml:Course
1: addCourse("UML")
2: addCourse("MDA")
1.1: «create»
:RegistrationManager
:Registrar
link
2.1: «create»
mda:Course
object creation message


iteration specifier
sd PrintCourses
iteration clause
1.1 * [for i = 1 to n] : printCourse(i)
1: printCourses ( )
:RegistrationManager
:Registrar
1.1.1: print()
[i]:Course

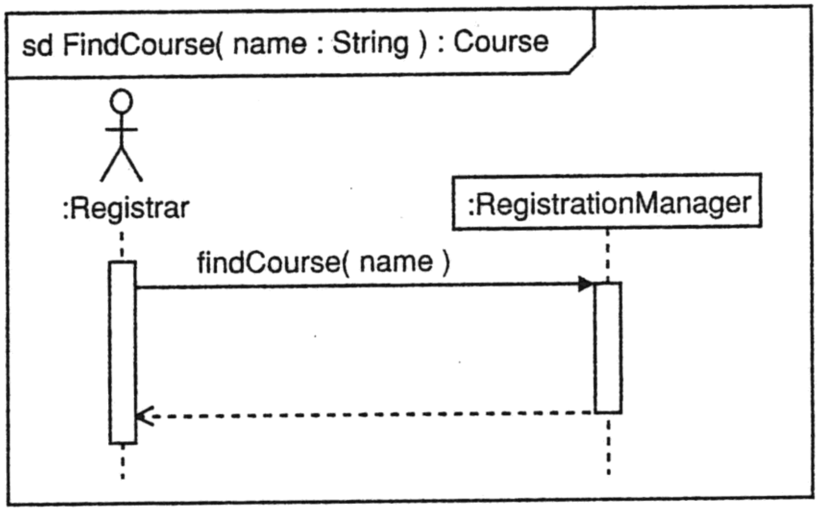
✓ Reusable interaction fragment

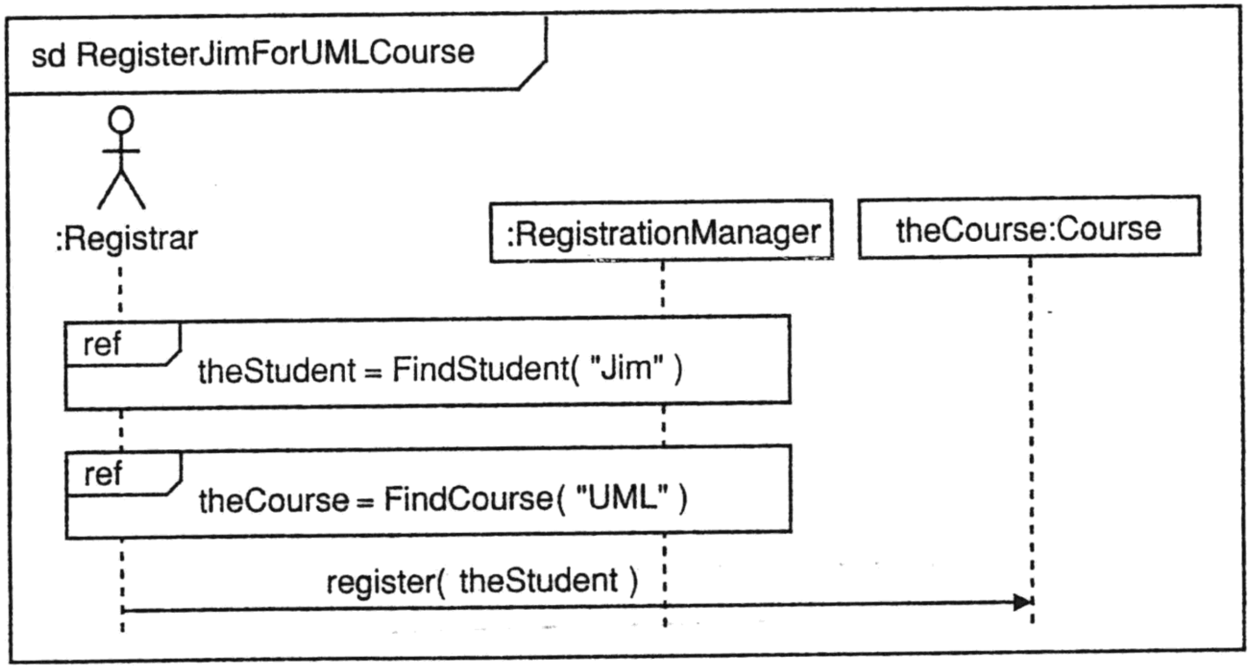

sd LogOnRegistrar
:Registrar
:SecurityManager
logOn(userName, password)
authenticate(userName, password)

self-delegation: a lifeline sends a message to itself, it is a nested activation

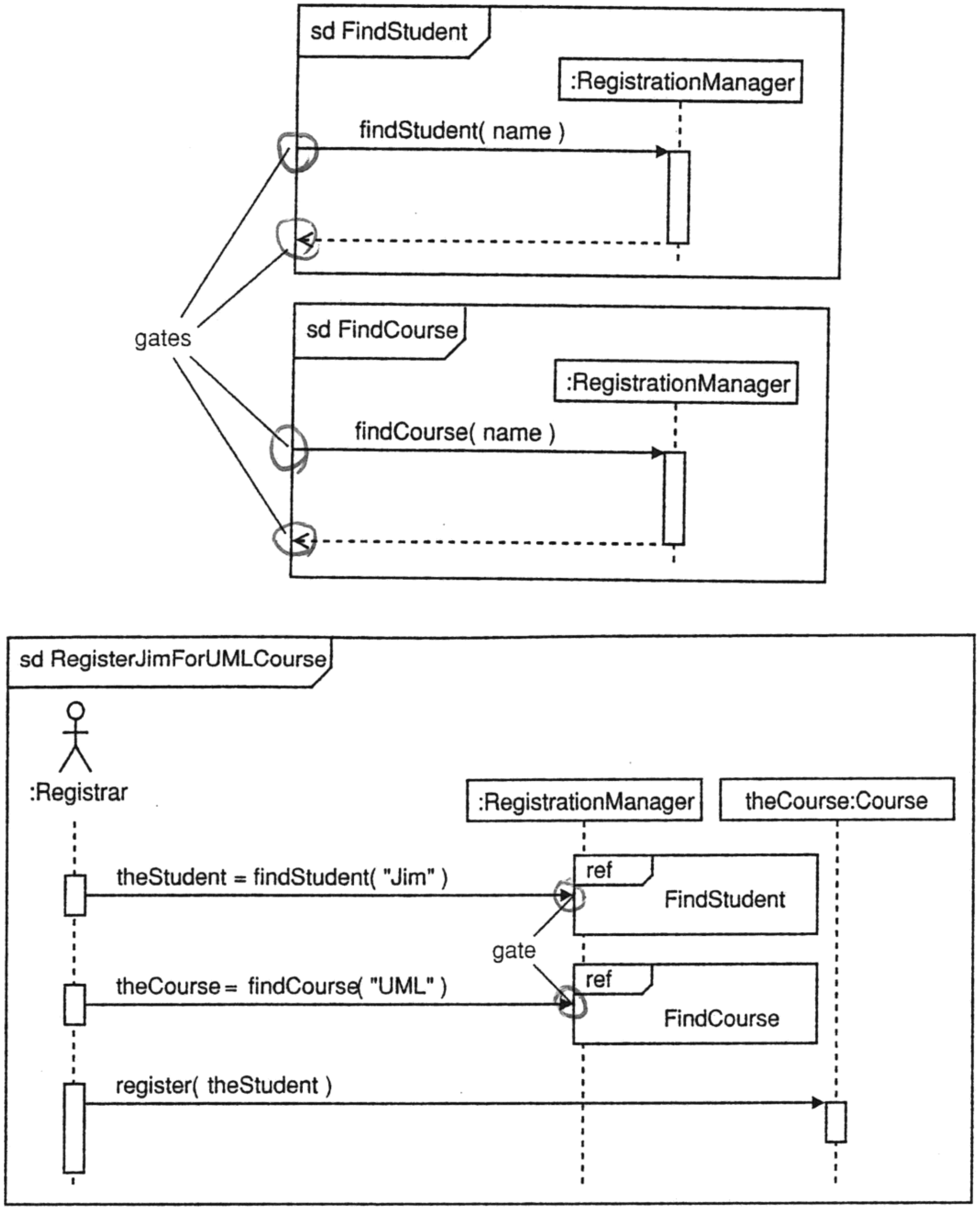


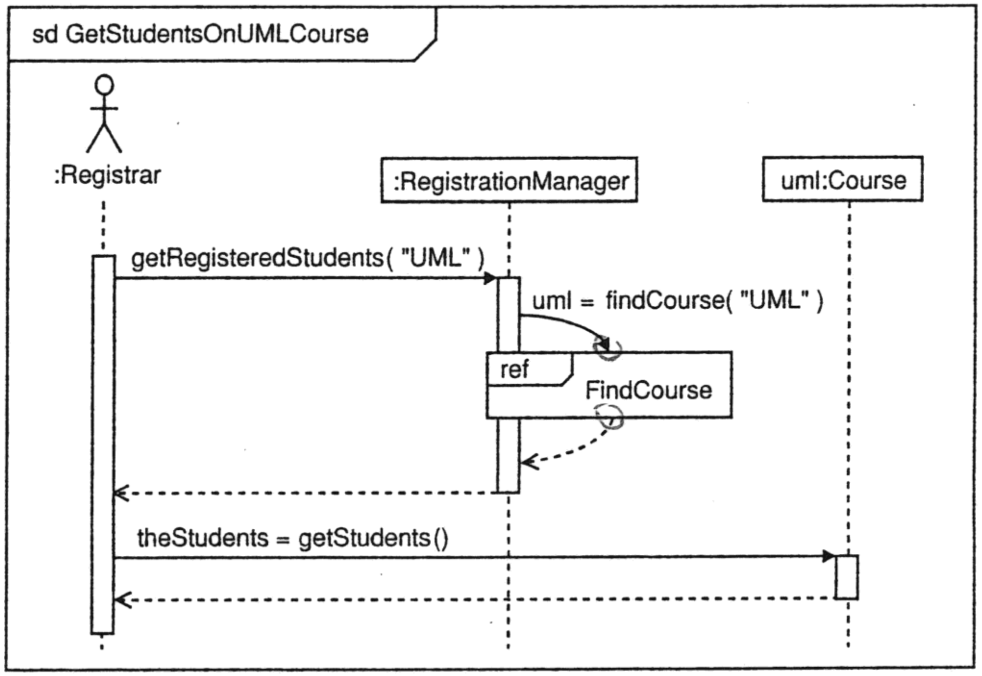
✓ Parameters in reusable interaction fragment



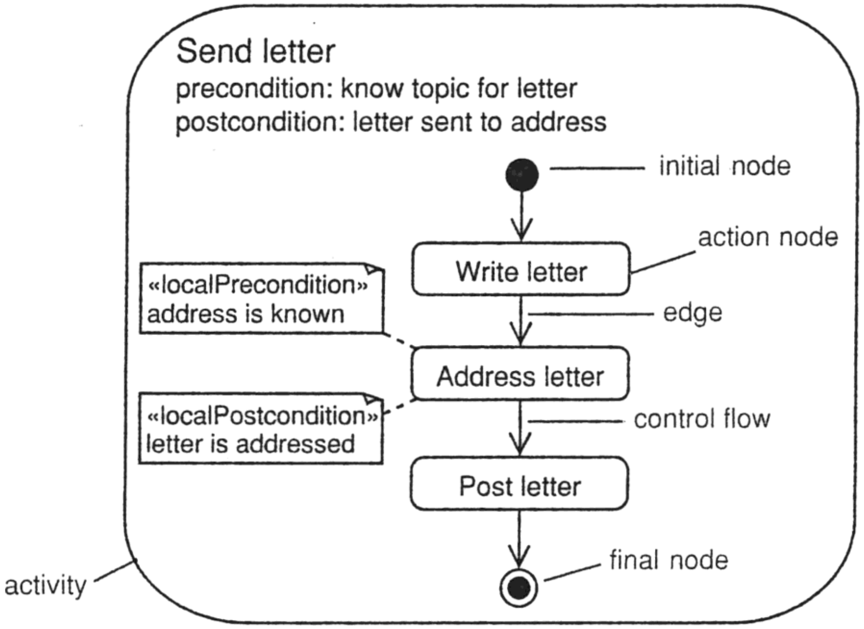

sd RegisterJimForUMLCourse
:Registrar
:RegistrationManager
theCourse:Course
ref
theStudent = FindStudent("Jim")
ref
theCourse = FindCourse("UML")
register(theStudent)

✓ Gates: inputs and outputs of interactions outside the frame

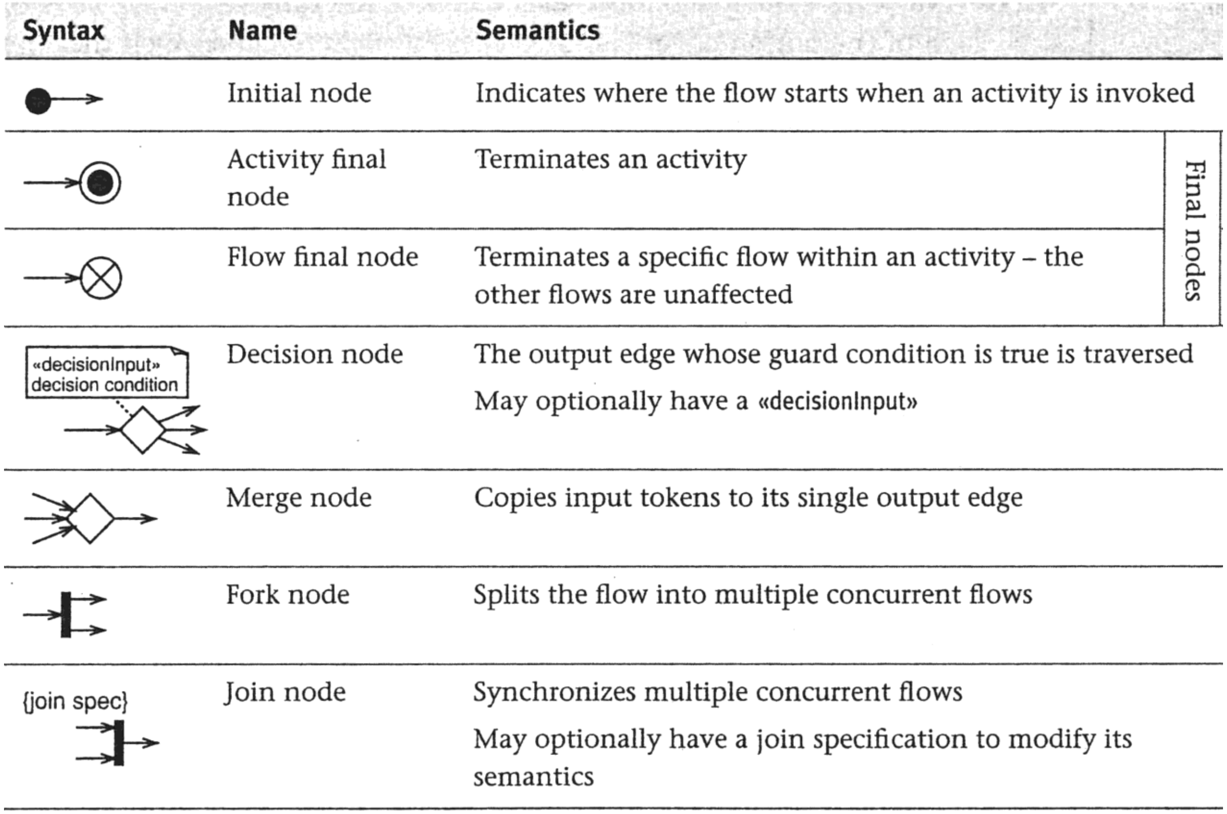



sd GetStudentsOnUMLCourse
:Registrar
:RegistrationManager
uml:Course
getRegisteredStudents("UML")
uml = findCourse("UML")
ref
FindCourse
theStudents = getStudents()

- ✓ Activity diagrams:

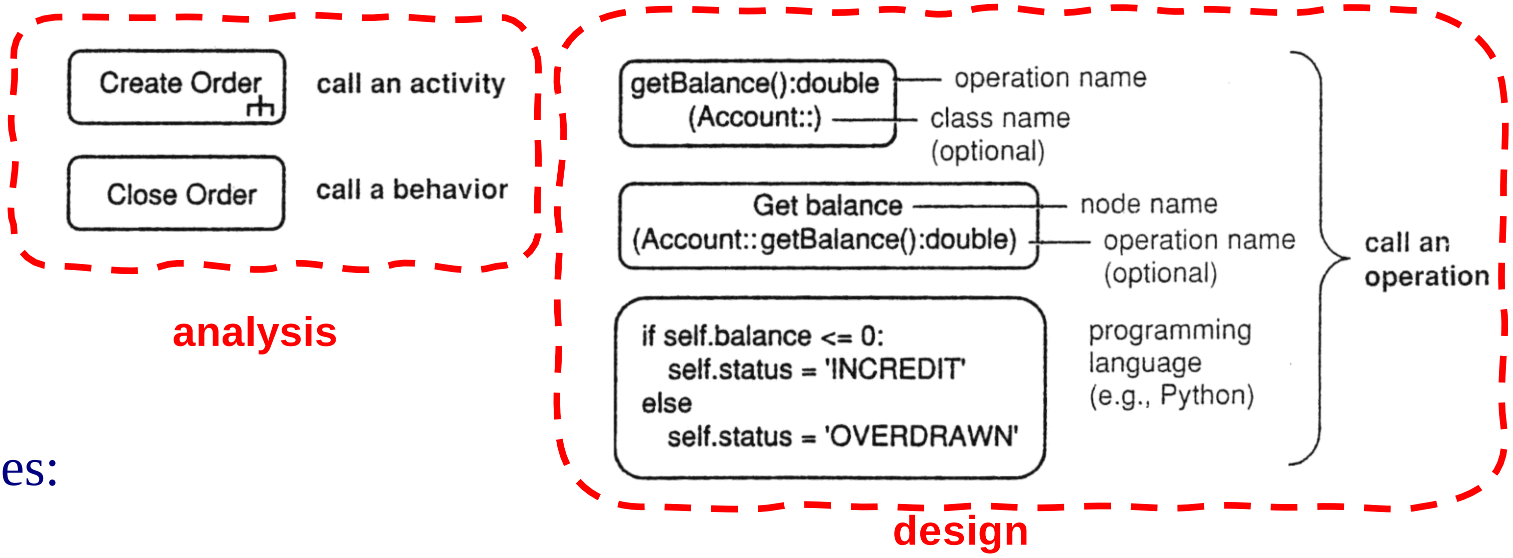


✓ Control nodes:

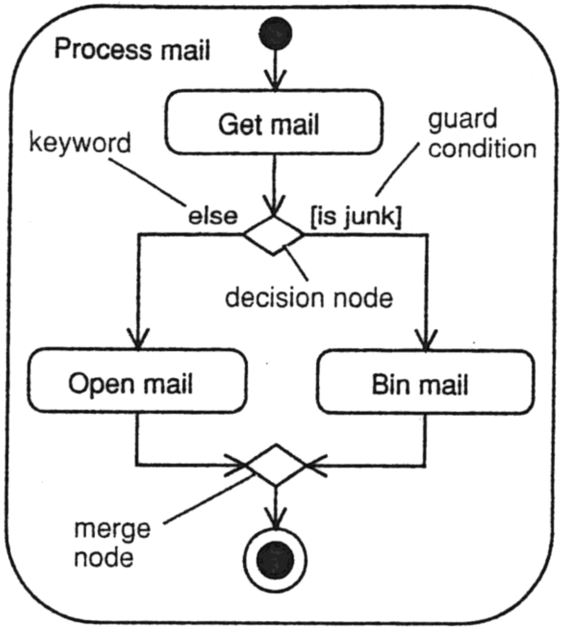


Syntax	Name	Semantics	
●→	Initial node	Indicates where the flow starts when an activity is invoked	
→◉	Activity final node	Terminates an activity	Final nodes
→⊗	Flow final node	Terminates a specific flow within an activity – the other flows are unaffected	
«decisionInput» decision condition	Decision node	The output edge whose guard condition is true is traversed May optionally have a «decisionInput»	
	Merge node	Copies input tokens to its single output edge	
	Fork node	Splits the flow into multiple concurrent flows	
{join spec}	Join node	Synchronizes multiple concurrent flows May optionally have a join specification to modify its semantics	

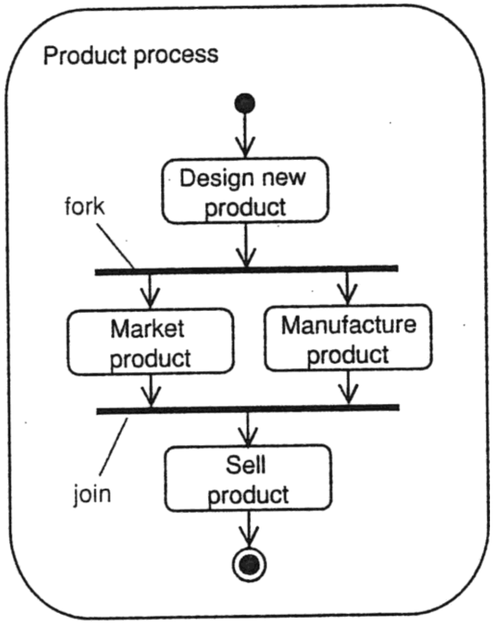
✓ Call action nodes:



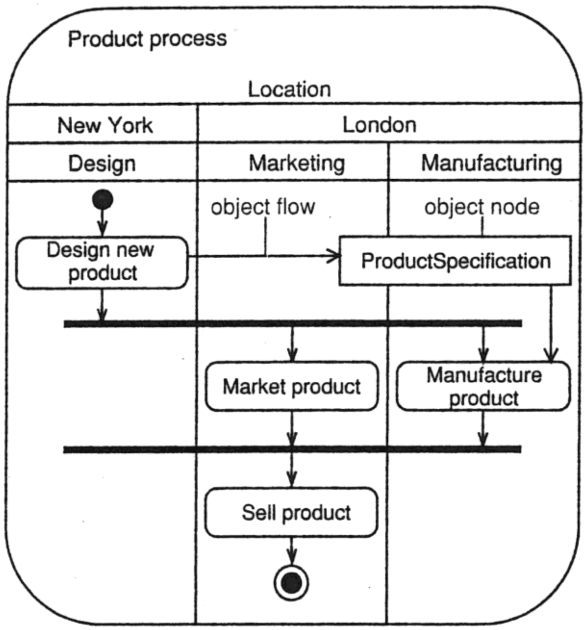
```
if self.balance <= 0:
   self.status = 'INCREDIT'
else
   self.status = 'OVERDRAWN'
```



decision/merge nodes

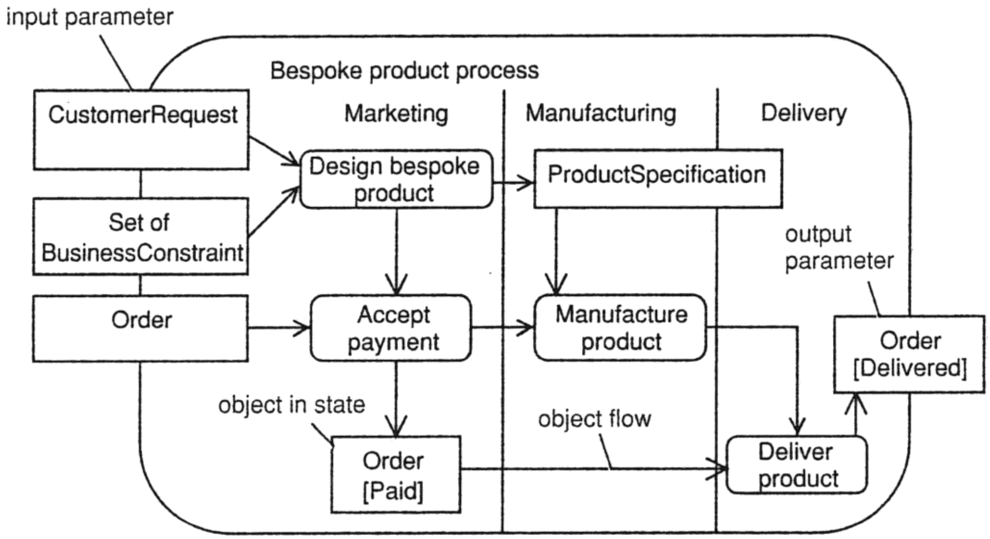


fork/join nodes

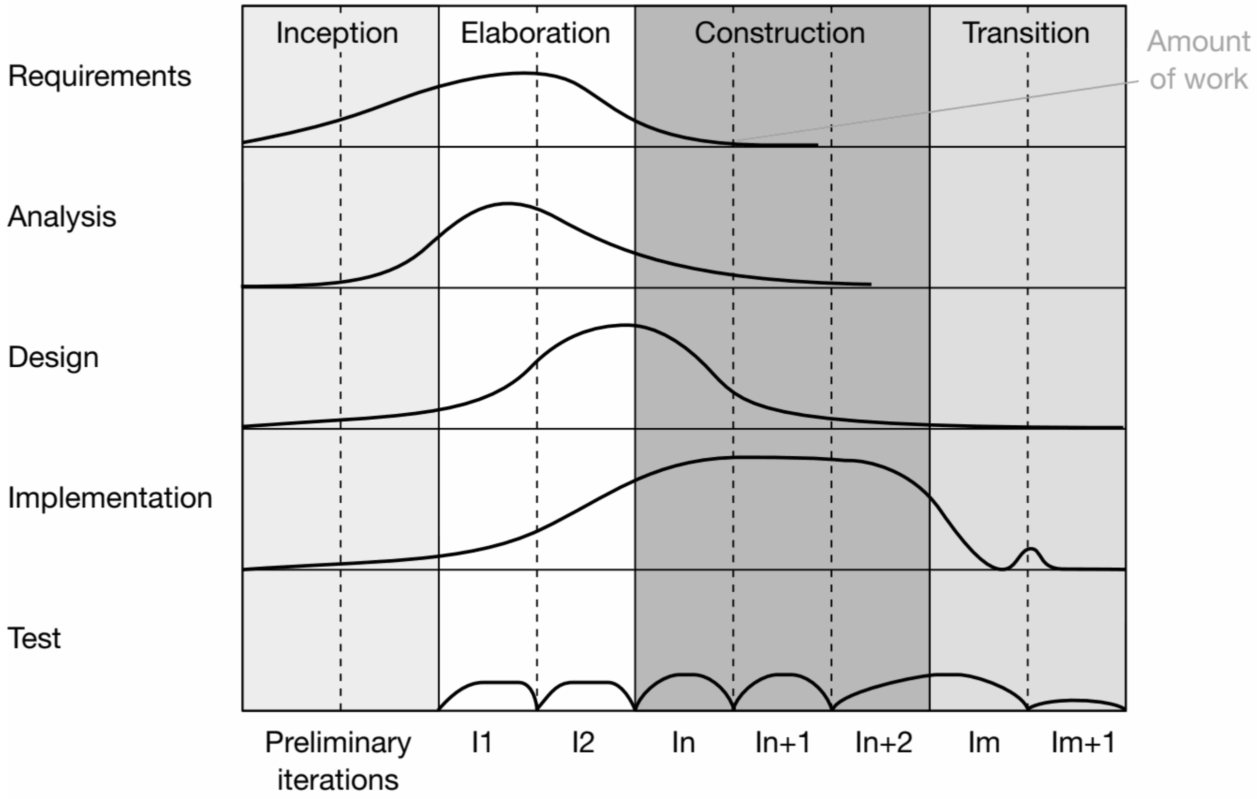


object nodes

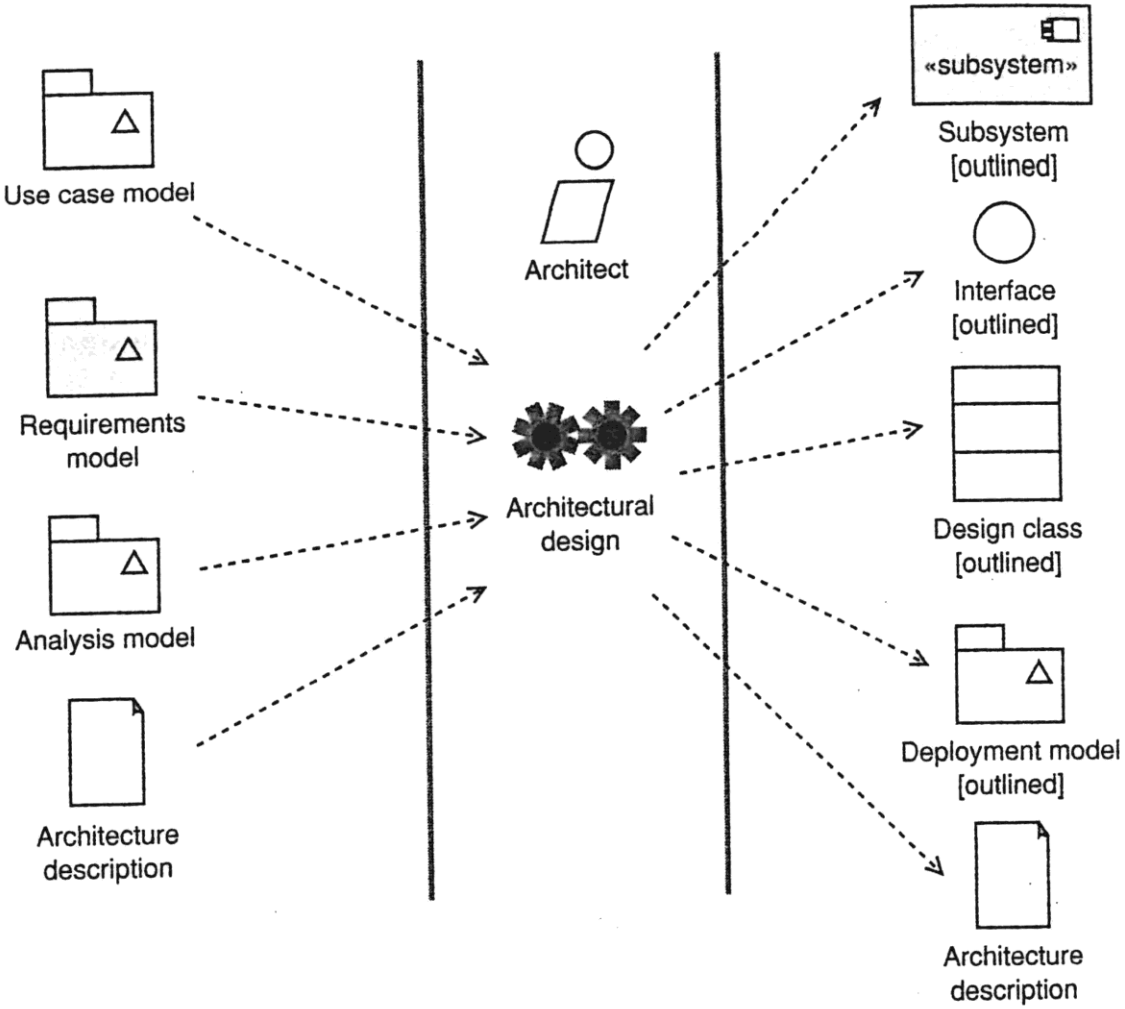
✓ I/O params and object in state:



3. The Design workflow

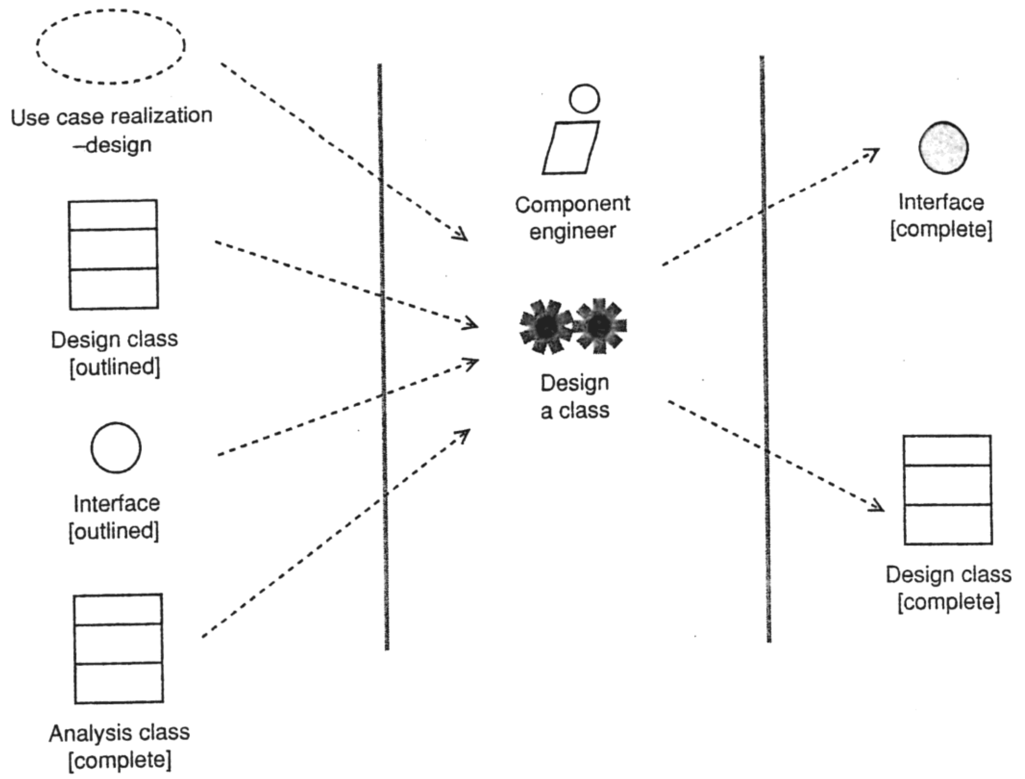


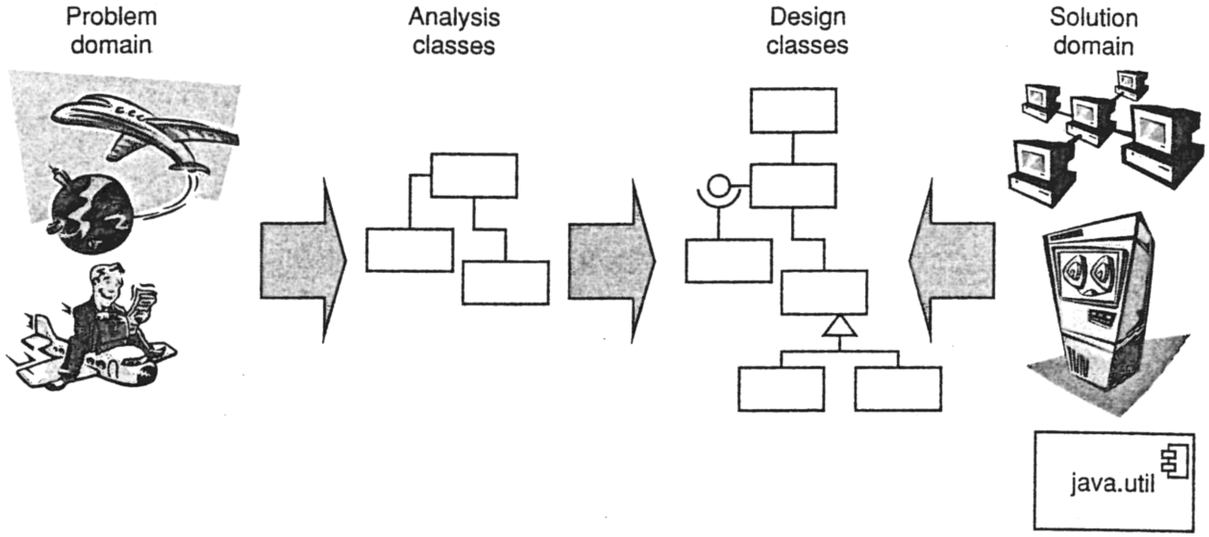
- ✓ While Requirements and Analysis workflows focus on the problem domain from the point of view of the system stakeholders, Design workflow focuses on the solution domain to provide: design subsystems, design classes, interfaces, use case realizations design, deployment diagrams.



- ✓ Design classes and interfaces are first outlined and the sufficiently detailed to serve as a good basis for creating source code

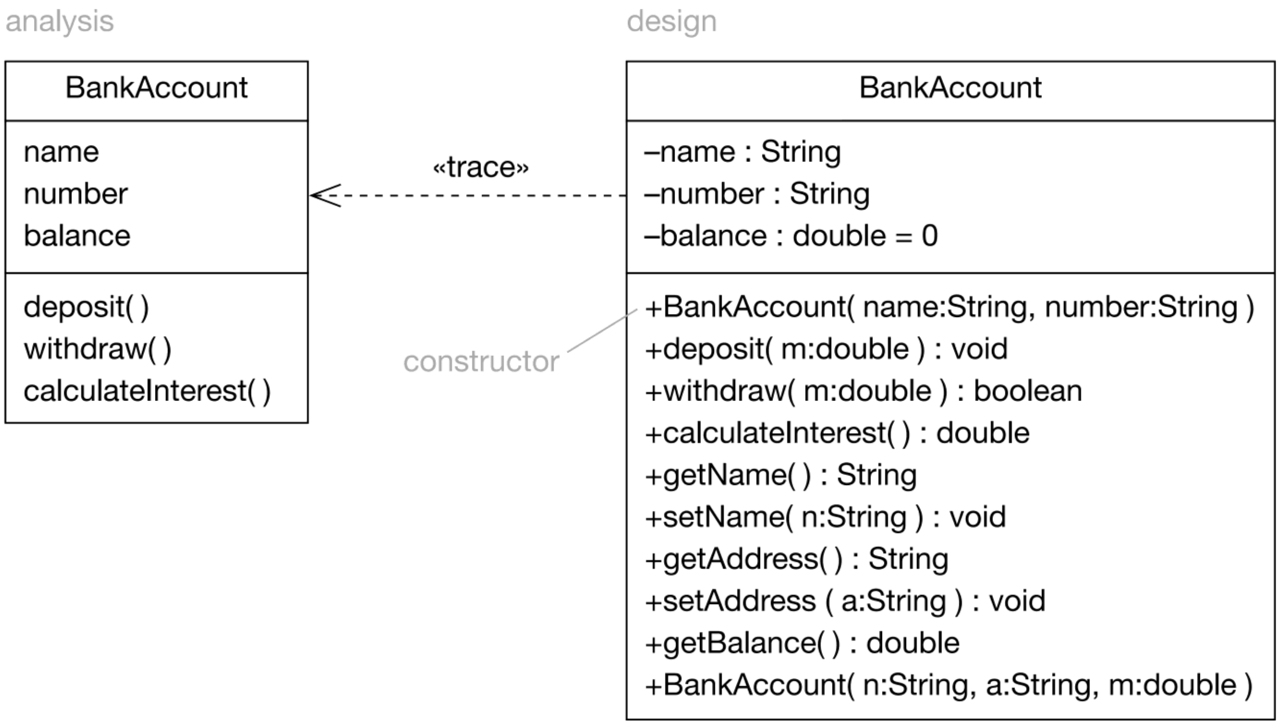
- ✓ Some design classes are refinements of analysis classes. Other design classes are based on the solution domain (e.g. utility classes, communication middleware, db)


Use case realization
–design
Design class
[outlined]
Interface
[outlined]
Analysis class
[complete]
Component
engineer
Design
a class
Interface
[complete]
Design class
[complete]


Problem
domain
Analysis
classes
Design
classes
Solution
domain
java.util

✓ Complete the set of attributes and fully specify them including name, type, visibility and (optionally) a default value.

✓ Turn the operations specified in the analysis class into a complete set of one or more methods.



analysis

BankAccount

name
number
balance

deposit( )
withdraw( )
calculateInterest( )

«trace»

constructor

design

BankAccount

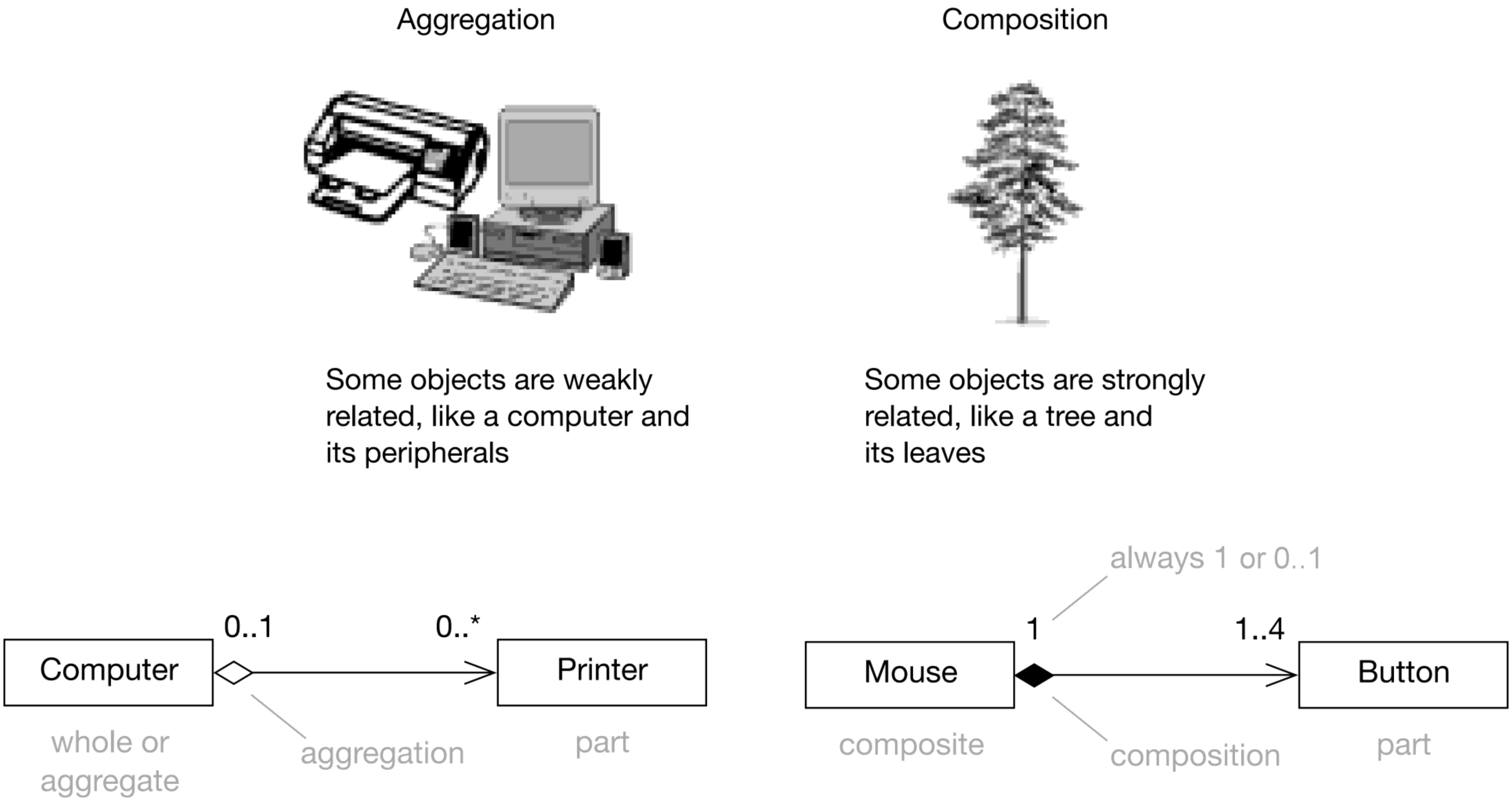
–name : String
–number : String
–balance : double = 0

+BankAccount(name:String, number:String)
+deposit(m:double) : void
+withdraw(m:double) : boolean
+calculateInterest( ) : double
+getName( ) : String
+setName(n:String) : void
+getAddress( ) : String
+setAddress (a:String) : void
+getBalance( ) : double
+BankAccount(n:String, a:String, m:double)

✓ A cohesive class has a small set of responsibilities that are closely related. Every operation, attribute, and association of the class is designed for the small, focused set of responsibilities.

- ✓ Operations offer a single primitive, atomic service. Do not offer multiple ways of doing the same thing, e.g. BankAccount class with operations for both single and multiple deposits (→maintenance and consistency problems).

- ✓ Refine analysis relationships: type, multiplicities, role names, navigability.



Aggregation

Composition

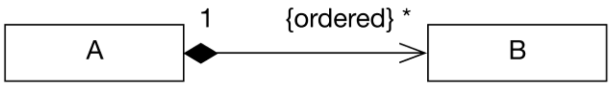
Some objects are weakly related, like a computer and its peripherals

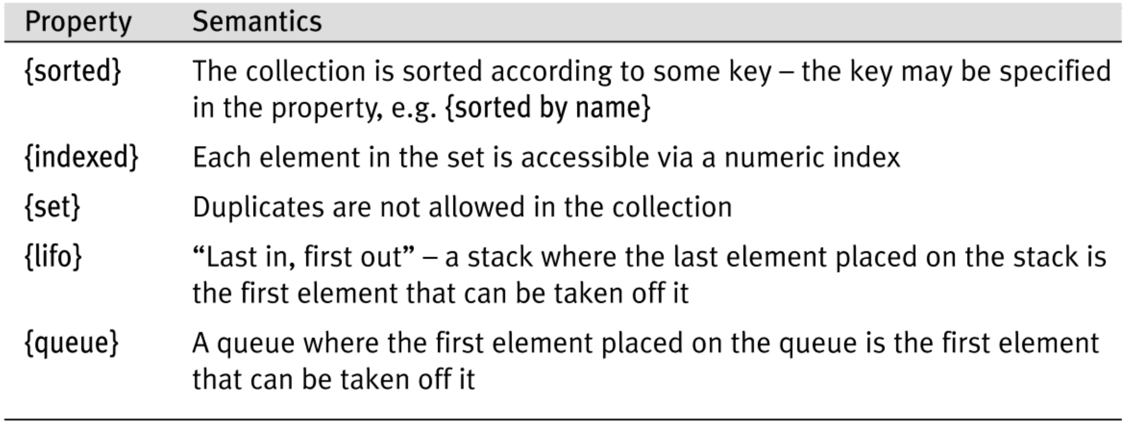
Some objects are strongly related, like a tree and its leaves

- ✓ The parts can exist (or not) independently of the aggregate, it is possible to share parts between aggregates.

- ✓ The parts can only belong to one composite at a time, no shared ownership; the composite has responsibility the creation/destruction or release of its parts.

- ✓ Multiplicity and constraints, semantics of collection (properties)

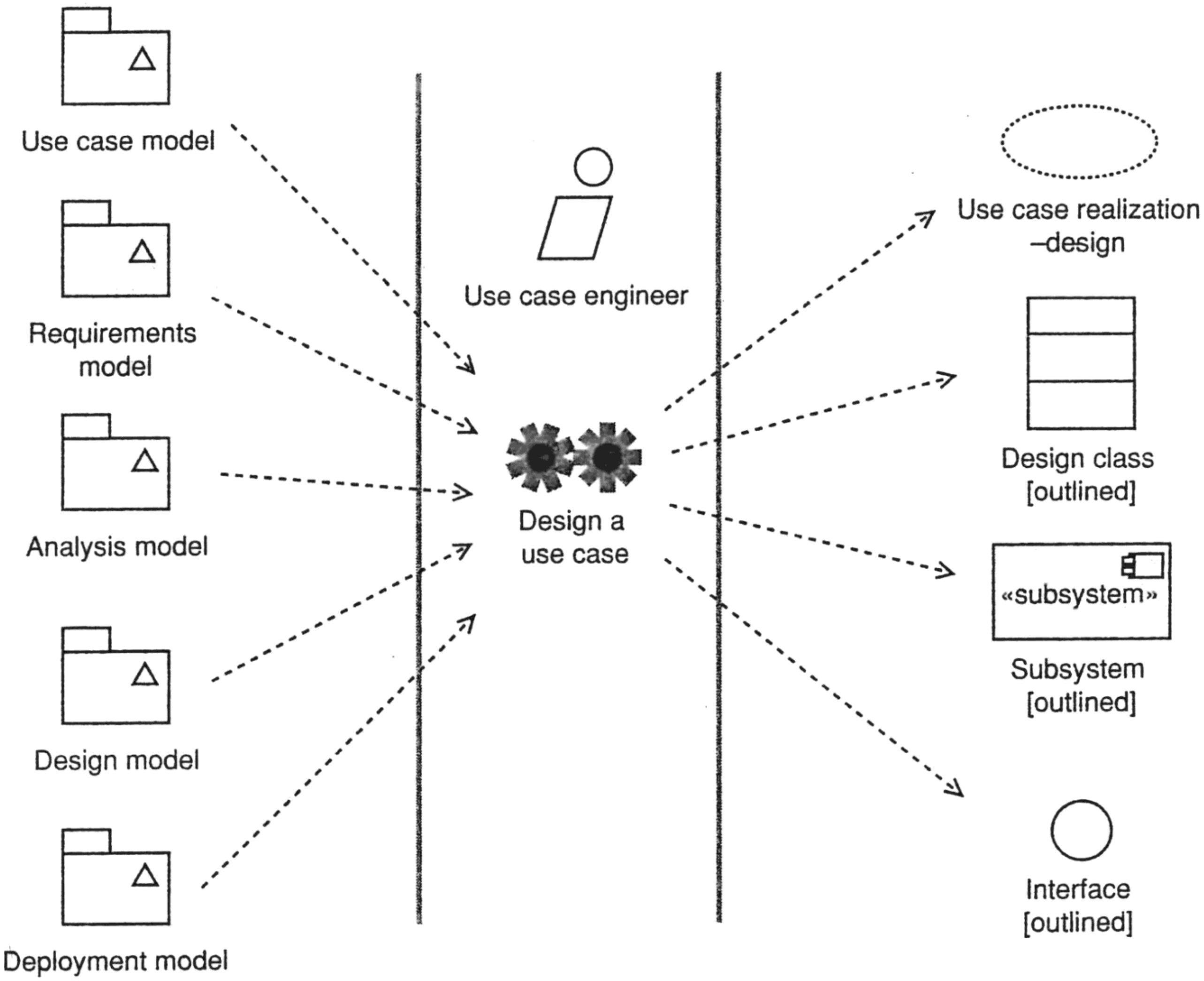




Property	Semantics
{sorted}	The collection is sorted according to some key – the key may be specified in the property, e.g. {sorted by name}
{indexed}	Each element in the set is accessible via a numeric index
{set}	Duplicates are not allowed in the collection
{lifo}	"Last in, first out" – a stack where the last element placed on the stack is the first element that can be taken off it
{queue}	A queue where the first element placed on the queue is the first element that can be taken off it

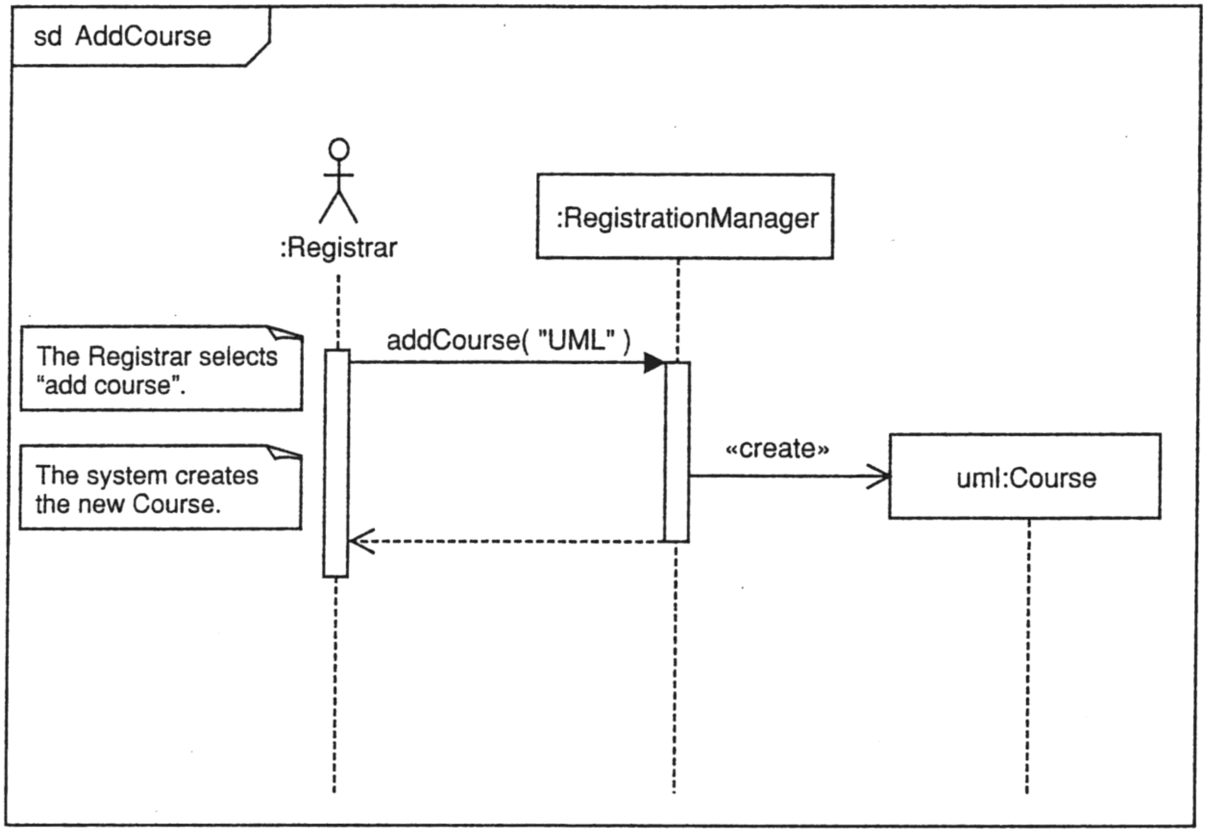
- ✓ Interfaces and components: breaking up the system into subsystems and determining their interactions via interfaces

- ✓ The activity “design a use case” is about finding design classes, interfaces, components that interact to provide the behavior specified by a use case.

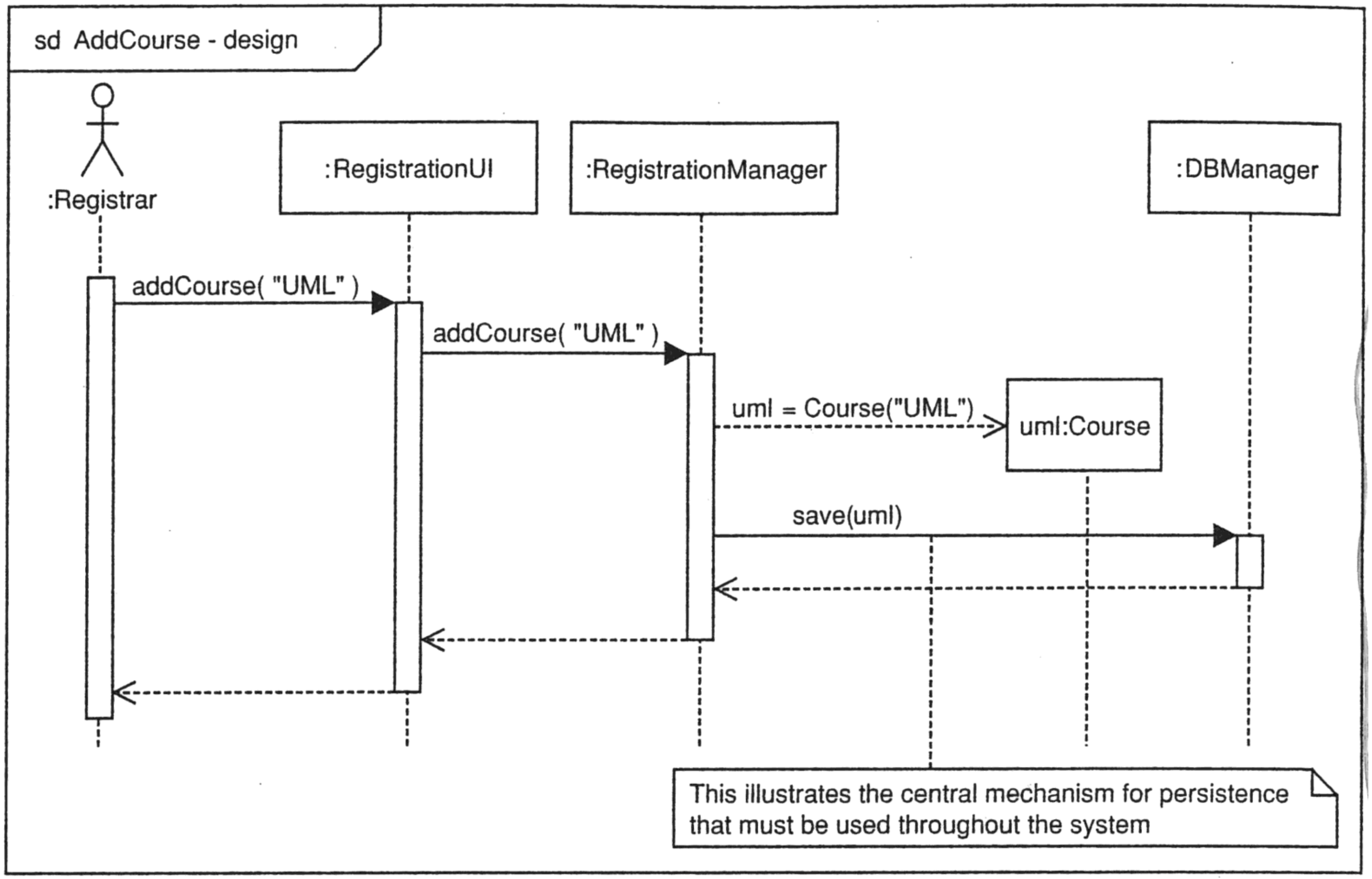


- ✓ Use case realization-design: design interaction diagrams and design class diagrams .

- ✓ Example of an analysis sequence diagram

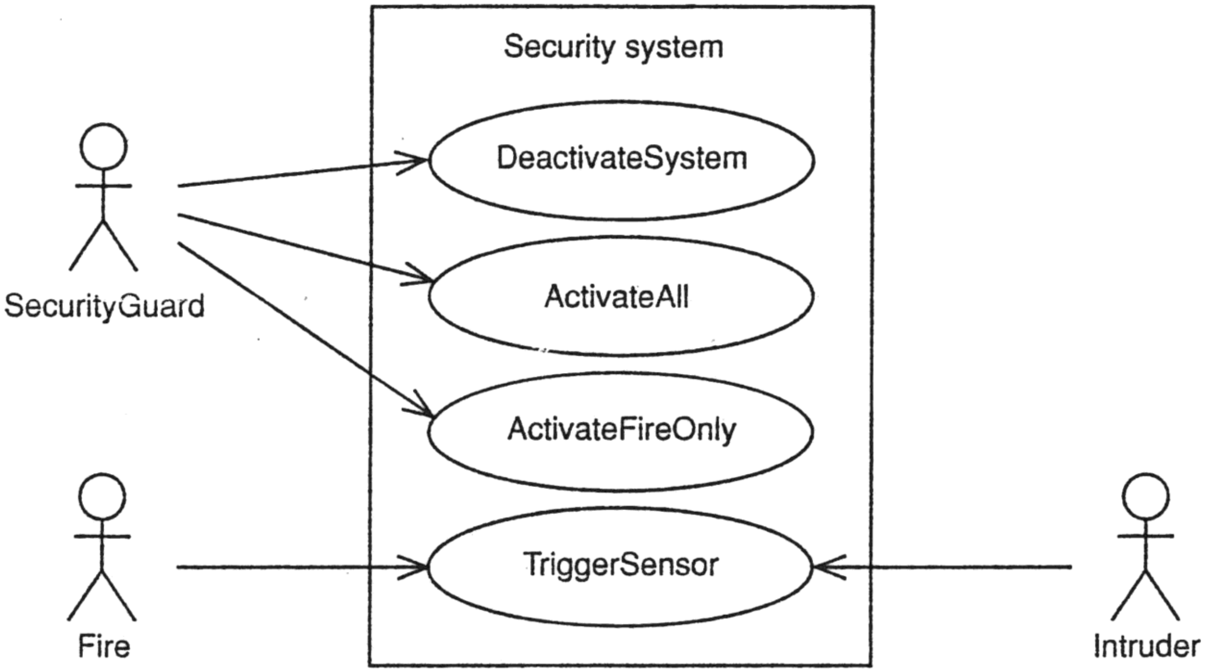


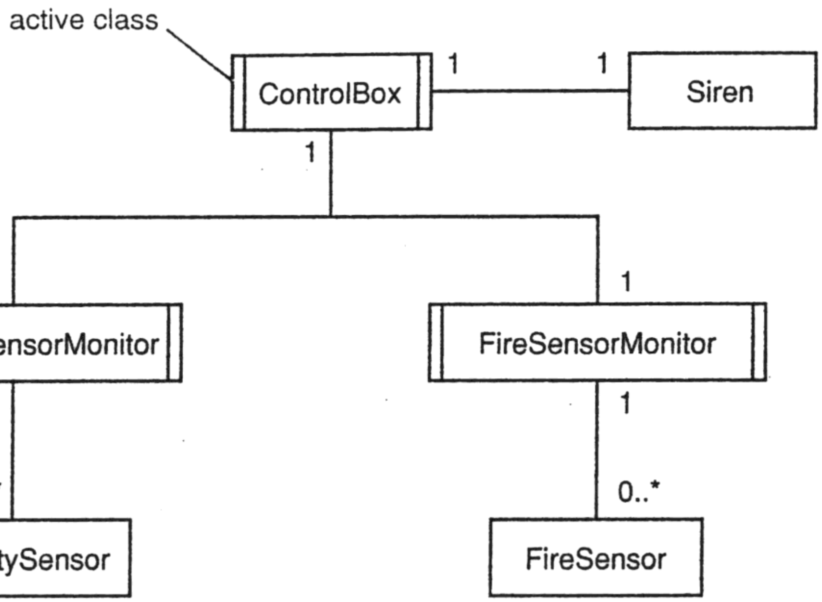
- ✓ In the corresponding design diagram, in the early stage of design, application layers are visible (e.g. front-end/GUI and backend/DB),

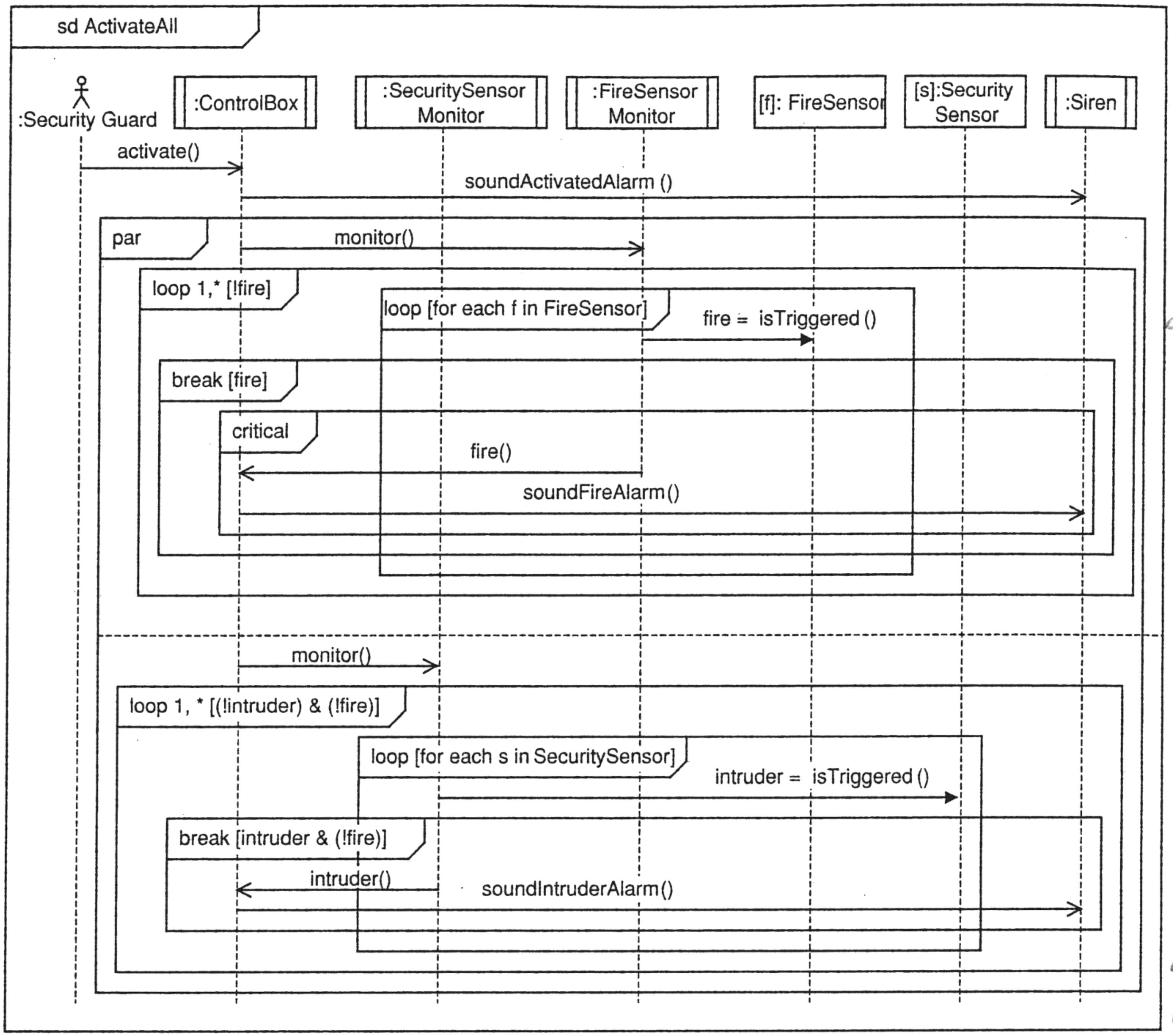


- ✓ Example of a security system realized with active class (its object encapsulates its own thread of control). It is made by four components: control box, siren, fire sensors, set of security sensors. There is a controller card for each type of sensor. The system is multithreaded.

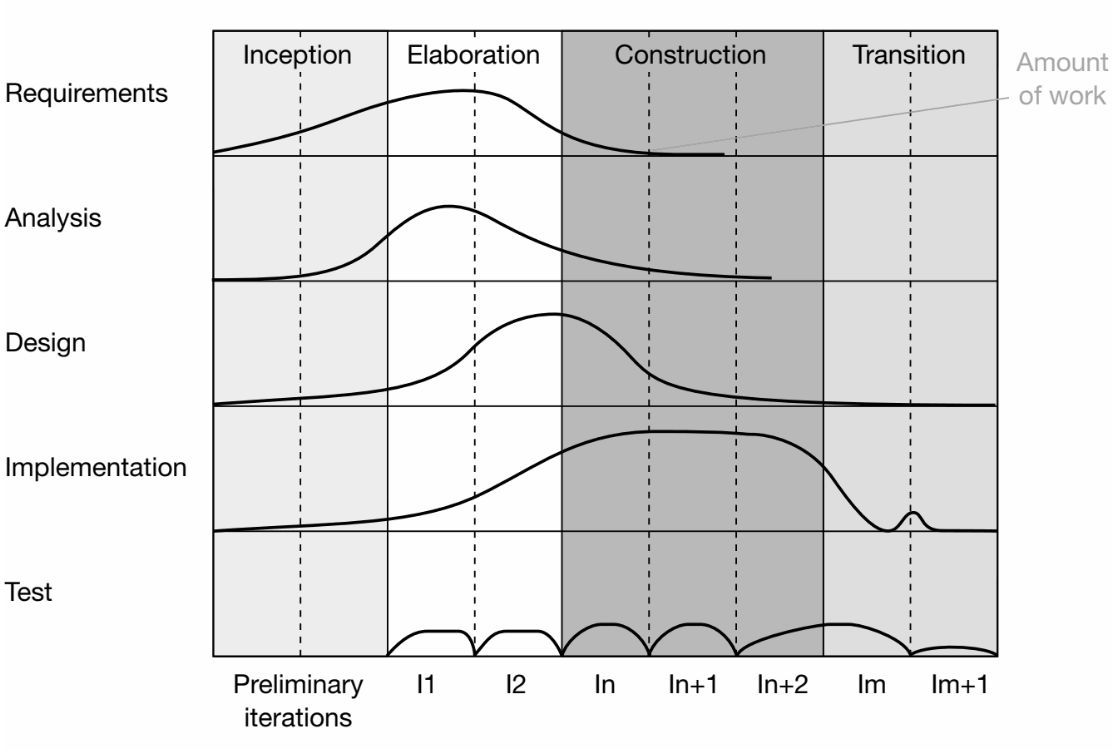
- ✓ Example of concurrency in sequence diagrams.


Security system
SecurityGuard
DeactivateSystem
ActivateAll
ActivateFireOnly
TriggerSensor
Fire
Intruder

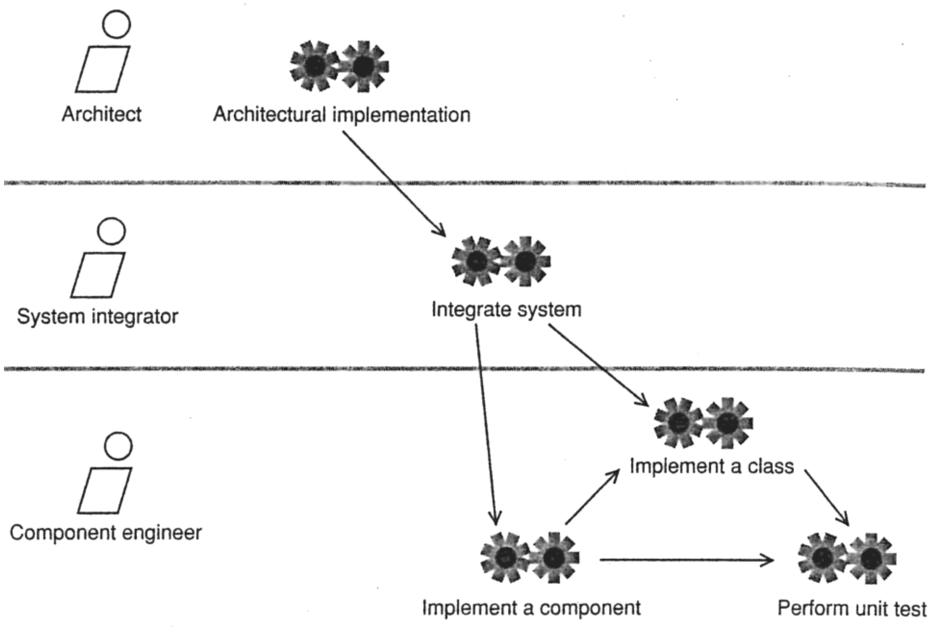

active class
ControlBox
1
1
Siren
1
1
SecuritySensorMonitor
1
0..*
SecuritySensor
1
FireSensorMonitor
1
0..*
FireSensor

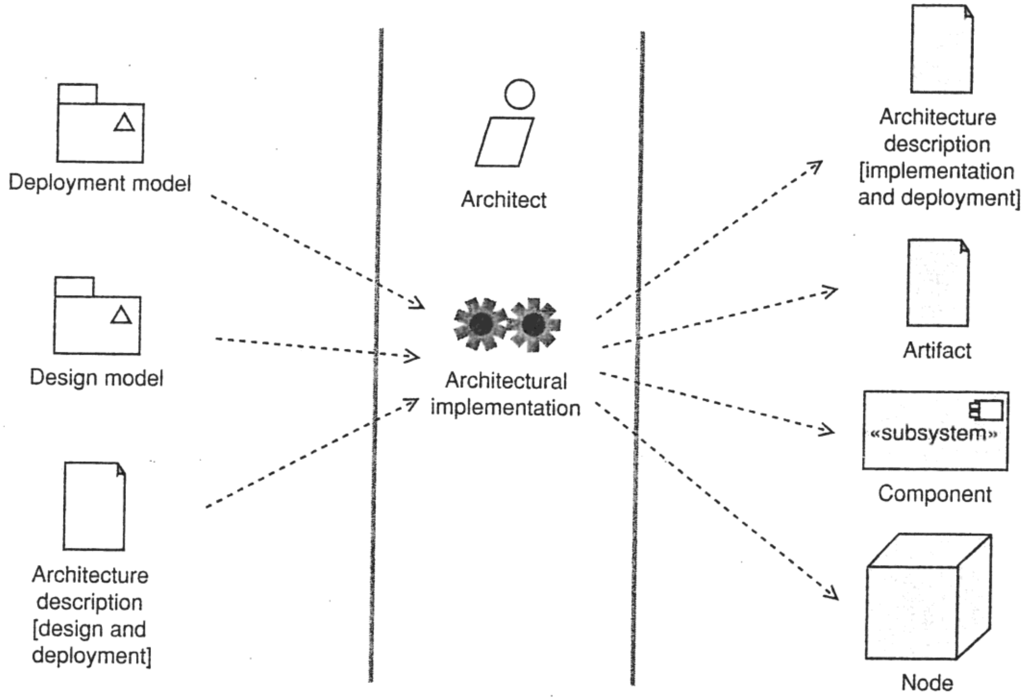

sd ActivateAll
:Security Guard
:ControlBox
:SecuritySensor Monitor
:FireSensor Monitor
[f]: FireSensor
[s]:Security Sensor
:Siren
activate()
soundActivatedAlarm ()
par
monitor()
loop 1,* [!fire]
loop [for each f in FireSensor]
fire = isTriggered ()
break [fire]
critical
fire()
soundFireAlarm()
monitor()
loop 1, * [(!intruder) & (!fire)]
loop [for each s in SecuritySensor]
intruder = isTriggered ()
break [intruder & (!fire)]
intruder()
soundIntruderAlarm()

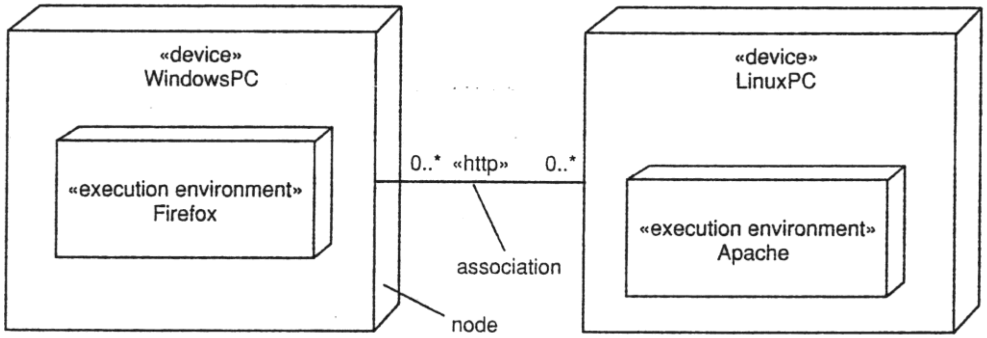
4. The Implementation workflow



- ✓ Implementation: to transform a design model into executable code
- ✓ It begins in the elaboration and is the main focus of the construction phase.
- ✓ Architectural implementation: to identify architecturally significant components and to map them to hw.


Architect
Architectural implementation
System integrator
Integrate system
Component engineer
Implement a class
Implement a component
Perform unit test


Deployment model
Design model
Architecture description [design and deployment]
Architect
Architectural implementation
Architecture description [implementation and deployment]
Artifact
«subsystem»
Component
Node



- ✓ Deployment diagram

  Device: a physical type of device (PC, Server)
  Execution environment: e.g. an Apache web server

- ✓ Types of artifacts: source files, executable files, scripts, database tables, documents, outputs of previous